RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITÉ DE MASCARA

FACULTÉ DES SCIENCES EXACTES

POLYCOPIÉ DE COURS

Programmation logique

BOUDIA Cherifa

Cours
Destinés aux étudiants en 3ème années licence Informatique
dans le

Département d'Informatique

2 juin 2022

Avant-propos

Pour les informaticiens, la connaissance des différentes familles de langages de programmation est nécessaire pour plusieurs raisons :

— La connaissance des différents concepts des langages de programmation permet de mieux exprimer vos propres idées lors du développement de logiciels.
— Une connaissance de base de divers langages de programmation est nécessaire afin de sélectionner le langage le plus approprié pour des projets spécifiques.
— C'est aussi le travail des informaticiens de concevoir de nouveaux langages de programmation.

Cela ne peut être fait que sur la base des langues qui ont déjà été développées.

Une distinction générale est faite entre les langages de programmation impératifs et déclaratifs (dans lesquels les langages déclaratifs sont subdivisés en langages fonctionnels et logiques). Dans les langages impératifs, les programmes sont composés d'une séquence d'instructions exécutées les unes après les autres, qui changent les valeurs des variables en mémoire. La plupart des langages de programmation utilisés aujourd'hui reposent sur ce principe, qui est également directement lié à l'architecture informatique classique qui remonte à John von Neumann.En programmation déclarative, par contre, les programmes consistent en une spécification de ce qui doit être calculé. C'est à l'interpréteur ou au compilateur de déterminer comment le calcul doit s'exécuter. Les langages de programmation déclaratifs sont donc orientés problèmes au lieu d'être orientés machine.

D'une part, les différents langages de programmation sont tous «égaux», c'est-à-dire que chaque programme peut en principe être écrit dans n'importe lequel des langages couramment utilisés. D'autre part, les langues sont différemment bien adaptées à différents domaines d'application. Par exemple, des langages impératifs tels que C sont utilisés pour une programmation rapide au niveau machine, car là le programmeur peut (et doit) prendre en charge la gestion de la mémoire. Dans d'autres langages de programmation, cette tâche est effectuée automatiquement (par le compilateur ou le système d'exécution). Cela permet un développement de programme plus rapide et moins sujet aux erreurs.

Alors qu'en programmation fonctionnelle un programme réalise une fonction (et aussi en programmation impérative un programme correspond implicitement à une fonction qui change les valeurs des variables), les programmes logiques consistent en des règles pour définir les relations (c'est-à-dire qu'un programme logique est une collection d'énoncés sur la relation entre différents objets). Lorsque le programme est exécuté, ces instructions sont utilisées pour répondre et résoudre des requêtes.

Nous considérerons le langage Prolog comme un exemple de programmation logique. Il est principalement utilisé en intelligence artificielle (par exemple pour les systèmes experts, le traitement du langage, les bases de données déductives, etc.). Ce qui suit explique brièvement les principes de base de Prolog pour motiver le contenu et la structure de ce cours.

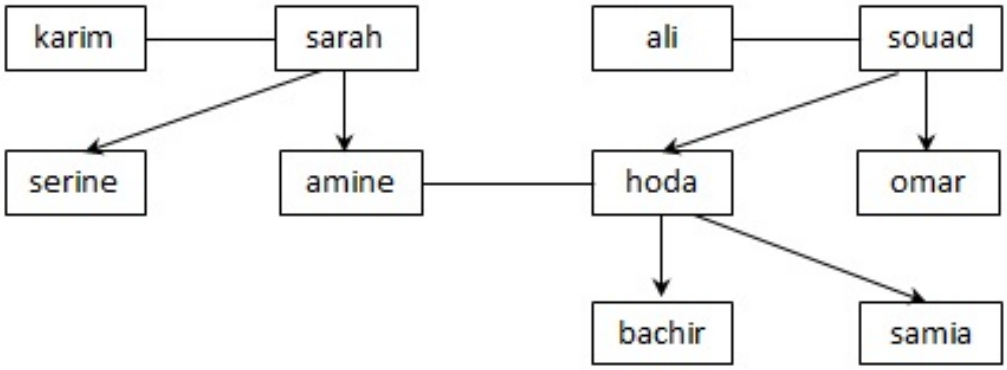
Table des matières

Chapitre 1

Introduction

1.1 Introduction

Le principe de la programmation logique repose sur le fait que le programmeur ne décrit que les connexions logiques d'un problème à résoudre, c'est-à-dire la base de connaissances. Aucune connaissance des détails de l'ordinateur au niveau de la machine n'est nécessaire pour cela LLOYD, 1984. À titre d'exemple, nous considérons la base de connaissances suivante sur les relations.



Les flèches vont des mères à leurs enfants et les lignes horizontales indiquent les personnes mariées.

1.2 Faits et demandes de renseignements

Le langage de la logique des prédicats est utilisé dans Prolog pour représenter cette base de connaissances. En fait, « **Prolog** » signifie « **Pro**gramming in **Log**ic » . La base de connaissances ne se compose que de formules logiques (dites **clauses**). Plus précisément, il existe deux types de clauses dans les programmes : les faits, qui font des déclarations sur les objets, et les règles, qui permettent d'inférer de nouveaux faits à partir de faits connus ROWE, 1988. Dans notre exemple, la base de connaissances se compose (initialement) uniquement de faits :

femelle(sarah).
femelle(serine).
femelle(souad).
femelle(hoda).
femelle(samia).
male(karim).
male(amine).
male(ali).

male(omar).
male(bachir).
marie(karim, sarah).
marie(ali, souad).
marie(amine, hoda).
mere(sarah, serine).
mere(sarah, amine).
mere(souad, hoda).
mere(souad, omar).
mere(hoda, samia).
mere(hoda, bachir).

Un fait se compose toujours du nom de la propriété ou de la relation (nommé symbole de prédicat) et immédiatement après (sans espaces) les objets qui ont cette propriété suivent puis finis par un point. Le fait suivant est sur la ligne suivante (ou du moins la précède d'un espace). Dans Prolog, les objets (comme **sarah**) et les propriétés (comme **mere**) commencent par des lettres minuscules. Le nombre d'arguments est appelé l'arité du symbole de prédicat. Un prédicat correspond à une fonction dont le résultat est « vrai » ou « faux ». Vous pouvez également voir que les relations sont dirigées. Donc, si un objet 1 est lié à l'objet 2, cela ne s'applique pas nécessairement à l'inverse NILSSON et MALUSZYNSKI, 1995. Par exemple, mere(sarah, serine) s'applique mais pas mere(serine, sarah). Les commentaires dans Prolog commencent par un **%** et vont à la fin de la ligne ou ils sont inclus dans **/ *** et ***** **/**.

L'exécution d'un programme logique signifie que l'utilisateur interroge le programme. C'est donc un langage de programmation « **conversationnel** ». Par exemple, une question possible serait : « **Est-ce un homme ?** ». Dans Prolog, les requêtes commencent par « **?-** ». ensuite viennent les énoncés (**formules**) à prouver. Vous pouvez également avoir plusieurs formules prouvées à la fois en séparant les formules par des virgules. À la fin de la question vient un point COLMERAUER, 1990. Dans prolog, on demanderait ceci

?- homme (ali).↵

Prolog génère la réponse en effectuant une preuve logique sur la base des connaissances existantes (c'est-à-dire sur la base de la description du problème donnée par le programmeur).

La tâche de l'ordinateur est donc de trouver la solution, c'est-à-dire que l'ordinateur est la machine d'inférence et « arithmétique » signifie « prouver » en Prolog. Ces preuves sont réalisées à l'aide des techniques d'unification et de résolution APT, 2001. L'ordinateur répondrait fidèlement à la requête ci-dessus :

?- marie(ali, sarah).

Cependant, la réponse est fausse. La calculateur suppose que sa base de connaissances contient toutes les connaissances pertinentes sur le monde. Donc, si une certaine affirmation ne découle pas de sa connaissance, elle est considérée comme fausse.

Pour cette raison, la déclaration est **marie(ali, sarah)** donc considéré comme faux. Afin de pouvoir envoyer des requêtes à un programme, le programme doit bien sûr être chargé au préalable. Si le programme est dans un fichier avec le nom « **file.pl** », entrez « **consult(file).** » (Ou « **[file].** » Pour faire court) dans l'interpréteur ou le compilateur Prolog. Les connaissances issues de cette base de connaissances sont alors

disponibles. Le fichier doit commencer par une lettre minuscule BRATKO, 2011.

1.3 Variables dans les programmes

La base de connaissances peut également contenir des variables. Les variables dans Prolog commencent par des majuscules ou un trait de soulignement. À titre d'exemple, nous ajoutons le fait suivant à la base de connaissances ci-dessus :

humain(X).

Les variables de la base de connaissances représentent toutes les affectations possibles, c'est-à-dire que ce fait signifie : « Tous les objets sont des humains ». Si vous faites maintenant la requête

?- humain(ali).

donc, comme prévu, la réponse est vraie. Cependant, la requête « ?- humain. » conduit au résultat vrai. Les mêmes variables dans les mêmes faits signifient l'égalité. Le fait « aime(X, Y) », signifie : « Tout le monde aime tout le monde ».

D'autre part, « aime(X, X) » signifie : « Tout le monde s'aime ». Cependant, les mêmes variables dans différentes clauses n'ont rien à voir les unes avec les autres.

1.4 Variables dans les requêtes

Il est également possible que le programme calcule lui-même les solutions. Pour ce faire, nous écrivons des variables dans les requêtes. À titre d'exemple, considérons la demande

?- mere(X, hoda).

Cela correspond à la question « Qui est la mère de Hoda ? » ou « Y a-t-il une affectation de la variable X pour que X soit la mère de hoda ? ».

L'ordinateur ne répond pas (seulement) par vrai, mais recherche une telle affectation de X.

La réponse est donc X = souad. Les variables de la base de connaissances sont donc toutes quantifiées et les variables des requêtes sont quantifiées d'existence. Comme autre exemple, considérons la requête

?- mere(souad, Y).

c'est-à-dire la question « Quels enfants Souad a-t-elle ? ». L'ordinateur répondrait maintenant par « Y = hoda ».

Mais ce n'est pas la seule solution possible. Si vous souhaitez calculer d'autres solutions, vous devez entrer un point-virgule. La calculateur recherche maintenant d'autres solutions et vous obtenez « Y = omar ». Donc tu peux voir que la mère n'est pas une fonction où l'entrée et la sortie sont fixes, mais une relation. Quelle est l'entrée et quelle est la sortie n'est pas fixe, cela dépend de la requête. Vous pouvez donc utiliser un seul et même programme, pour faire calculer tous ses enfants pour une femme et calculer la mère pour un enfant. Bien sûr, vous pouvez aussi demander « ?- mere(X, Y). » . C'est donc à l'utilisateur du programme quelles valeurs il spécifie

dans la requête et quelles valeurs il souhaite que le programme calcule.

Pour traiter ces demandes, Prolog recherche la base de connaissances de l'avant vers l'arrière et fournit la première réponse qui convient. Les clauses de la base de connaissances sont donc traitées de haut en bas BRAMER, 2013.

Comme autre exemple, considérons la requête « ?- humain(X). ». Toute instanciation possible de la variable X serait désormais une solution. La calculateur calcule ici la solution « la plus générale ». La raison en est que l'instruction est vraie pour chaque instanciation de X. Le programme essaie toujours de trouver les réponses les plus générales aux requêtes LUGER et STUBBLEFIELD, 2008.

1.5 Combinaison de questions

Comme déjà mentionné, vous pouvez faire prouver plusieurs affirmations en même temps (c'est-à-dire que vous pouvez combiner des questions).Un exemple est la requête suivante :

?- marie(ali, F), mere(F, hoda).

La question est donc de savoir s'il y a une femme F qui est mariée à ali et qui est aussi la mère de Hoda.

La variable F doit être instanciée dans l'ensemble de la requête. La procédure de Prolog est d'abord de résoudre le premier objectif « marie(ali F) ». On trouve une solution pour F. Avec cette solution on tente de résoudre « mere(F, hoda) ». Si cela ne fonctionne pas, on revient en arrière (retour en arrière) et on essaie de trouver une autre solution pour F qui satisfasse également « marie(ali F) ». Les instructions d'une requête sont traitées de gauche à droite. Comme autre exemple, considérons la question de savoir qui est la grand-mère de Samia :

?- mere(GrandMere, Maman), mere(Maman, samia).

Ici, GrandMere = sarah, Maman = serine est trouvée en premier.

Puis on recule jusqu'à ce qu'on trouve enfin la solution GrandMere = souad, Maman = hoda. Si, en revanche, les deux questions mere(GrandMere, Maman) et mere(Maman, samia) avaient été échangées, la solution aurait été trouvée plus rapidement (sans avoir à réinitialiser). En plus des faits, la base de connaissances peut également contenir des règles. Les règles servent à tirer de nouvelles connaissances à partir de connaissances connues ROWE, 1988. À titre d'exemple, considérons la relation père-enfant. Cette relation pourrait bien sûr être définie spécifiquement pour tous les objets, mais il est beaucoup plus court et plus clair de la formuler avec une règle générale. (En particulier, cela ne serait pas possible d'une autre manière avec un nombre infini d'objets ou avec un ensemble d'objets plus tardif ou décroissant.) La règle suivante stipule : « Une personne V est le père d'un enfant K s'il est marié à une femme F qui est la mère de l'enfant K »

pere(V, K) :- marie(V, F), mere(F, K).

Le signe « :- » signifie « si » et les règles formulent des relations « si-alors ». Si les hypothèses du côté droit de la règle sont vraies, alors la déclaration du côté gauche est également vraie. Le côté gauche s'appelle la tête de la règle et le côté droit s'appelle le tronc. Les exigences dans le corps sont séparées par des virgules et à la fin il y a

un point. La signification d'une règle p :- q, r. est :

Si q et r tiennent, alors p l'est aussi.

Lors du traitement (c'est-à-dire pour les preuves) dans Prolog, les règles sont appliquées à l'envers. Afin de montrer que le côté gauche d'une règle tient, il faut montrer que le côté droit tient (chaînage vers l'arrière). À titre d'exemple, considérons la demande

?- pere(ali, hoda).

Afin de prouver cette affirmation, il faut trouver quelque chose basé sur la règle de « pere », de sorte que marie (ali, F), mere(F, susanne) soient vraies. Cela correspond donc à la requête ?- marie(ali, F), mere(F, hoda). Prolog répond donc par « vrai ». De même, la requête serait

?- pere(ali, Y).

Les réponses sont Y = hoda et Y = omar.

Plusieurs règles pour un prédicat jusqu'à présent, nous avons défini une règle dans laquelle la tête tient si la conjonction des hypothèses est vraie. Considérons maintenant aussi le cas où la tête découle de la disjonction de deux hypothèses. Pour cela, nous utilisons plusieurs règles pour le même symbole de prédicat. À titre d'exemple, considérons un parent prédicat, où parent(X, Y) doit être vrai si X est la mère ou le père de Y. Les règles pour cela sont les suivantes :

parent(X, Y) :- mere(X, Y).
parent(X, Y) :- pere(X, Y).

Si vous faites maintenant la demande ?- parent(X, hoda).

Il en résulte les réponses X = souad et X = ali. On voit que l'ordre des clauses influe également sur la recherche et l'ordre des solutions.

1.6 Règles récursives

La récursivité est une technique de programmation importante dans Prolog. A titre d'exemple, nous définissons un ancêtre comme prédicat BRATKO, 2011. Ici, V est un ancêtre de X si V est un parent de X ou s'il existe un Y tel que V est le parent de Y (c'est-à-dire que V à un enfant Y) et Y est l'ancêtre de X. La traduction de cette règle en Prolog donne :

ancetre(V, X) :- parent(V, X).
ancetre(V, X) :- parent(V, Y), ancetre(Y, X).

La requête ?- ancetre(X, samia).

Cela signifie maintenant : « Qui sont les ancêtres de Samia ? ». Prolog y trouvera les réponses suivantes :

X = hoda ;
X = amine ;
X = sarah ;
X = souad ;

X = karim ;
X = ali

1.7 Caractéristiques des programmes logiques

Dans l'ensemble, les propriétés suivantes des programmes logiques résultent LLOYD, 1984 :

1. Les programmes (purement) logiques n'ont pas de structure de contrôle pour contrôler la séquence du programme. Les programmes ne sont que des ensembles de faits et de règles qui sont traités de haut en bas (ou de gauche à droite).
2. La programmation logique a émergé de la vérification automatique et lorsqu'un programme logique est exécuté, une tentative est faite pour prouver une requête. Les solutions pour les variables de la requête sont également calculées. Cela signifie que les variables d'entrée et de sortie ne sont pas fixées dans un programme logique.
3. Les programmes logiques sont particulièrement adaptés aux applications de l'intelligence artificielle. Par exemple, des systèmes experts peuvent être très bien mis en œuvre avec cela, les règles du programme étant formées à partir des connaissances des experts. Les autres domaines d'application principaux sont les bases de données déductives et le prototypage rapide.

La structure du cours est la suivante : Étant donné que les programmes logiques sont constitués de formules logiques de prédicat et que des preuves de logique de prédicat sont effectuées pour l'exécution de programmes logiques, les bases nécessaires de la logique de prédicat sont présentées au chapitre 2 et au chapitre 3, nous introduisons le principe de preuve de résolution utilisé dans la programmation logique. Dans le chapitre 4, nous examinons ensuite la syntaxe, la sémantique et l'expressivité des programmes de logique (pure) comme ci-dessus et discutons de la stratégie d'exécution des programmes de logique. Enfin, dans le chapitre 5, nous abordons ensuite l'arithmétique dans le langage de programmation Prolog et les opérateurs.

Chapitre 2

Bases de la logique des prédicats

2.1 Introduction

Dans ce chapitre, nous allons introduire le langage de la logique des prédicats du premier ordre, qui est utilisé pour la formulation de programmes logiques. Pour cela, nous récapitulons la syntaxe et la sémantique de la logique des prédicats dans les sections 2.2 et 2.3, notamment afin d'introduire les notations utilisées dans la suite. Pour développer un programme logique, il faut examiner si une formule (la requête) découle d'un ensemble de formules (les faits et les règles du programme).

2.2 Syntaxe de la logique des prédicats

La syntaxe définit de quelle série de caractères se composent les expressions d'un langage ZOMBORI, URBAN et BROWN, 2020. Nous définissons d'abord l'alphabet à partir duquel les expressions de la logique des prédicats sont formées.

2.2.1 Définition (signature)

Une signature (Σ, Δ) est un couple avec $\Sigma = \bigcup_{n\in\mathbb{N}} \Sigma_n$ et $\Delta = \bigcup_{n\in\mathbb{N}} \Delta_n$. Ici Σ et Δ sont l'union d'ensembles disjoints par paires Σ_n et Δ_n. Chaque $f \in \Sigma_n$ est le symbole de la fonction à n arguments et chaque $p \in \Delta_n$ est le symbole de prédicat à n arguments. Les éléments de Σ_0 sont également appelés constantes. Nous demandons ici toujours $\Sigma_n = \emptyset$ LLOYD, 1984.

2.2.2 Exemple

Nous considérons comme exemple la signature suivante (Σ, Δ) avec $\Sigma = \Sigma_0 \cup \Sigma_3$ et $\Delta = \Delta_1 \cup \Delta_2$. Il correspond à la signature du programme logique du chapitre 1, où Σ contient également le symbole de fonction à trois arguments date et Δ le symbole de prédicat à deux arguments.

$\Sigma_0 = \mathbb{N} \cup \{\text{sarah, serine, souad, hoda, samia, karim, amine, ali, omar, bachir}\}$
$\Sigma_3 = \{\text{date}\}$
$\Delta_1 = \{\text{femelle, male, humain}\}$
$\Delta_2 = \{\text{marie, mere, pere, parent, ancetre, ne}\}$

Nous définissons maintenant comment les objets de données sont représentés dans le langage de la logique des prédicats.

2.2.3 Définition (termes)

Soit (Σ, Δ) une signature et $\mathcal{V}$ ensemble de variables telles que $\mathcal{V} \cap \Sigma = \emptyset$ est vrai. Puis désigné $\mathcal{T}(\Sigma, \mathcal{V})$ est l'ensemble de tous les termes (sur Σ et $\mathcal{V}$) SHOHAM, 2014. Voici $\mathcal{T}(\Sigma, \mathcal{V})$ est le plus petit ensemble avec

— $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ et
— $f(t_1, ..., t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$, si $f \in \Sigma_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ pour certains n $\in \mathbb{N}$.$\mathcal{T}(\Sigma)$ représente $\mathcal{T}(\Sigma, \emptyset)$, c'est-à-dire pour l'ensemble de tous les termes sans variable (ou termes fondamentaux). Pour un terme t, $\mathcal{V}$ (t) est l'ensemble de toutes les variables de t.

Afin de pouvoir distinguer les variables des symboles de fonction et de prédicat (en particulier des constantes), nous utilisons (comme dans **Prolog**) les conventions selon lesquelles les variables commencent par des lettres majuscules et les symboles de fonction et de prédicat par des lettres minuscules.

2.2.4 Exemple

Reprenons la signature Σ de l'exemple 2.2.2. Si $\mathcal{V} = \{X, Y, Maman, GrandMere, ...\}$ alors nous obtenons par exemple, les termes suivants dans $\mathcal{T}(\Sigma, \mathcal{V})$: sarah, 42, date(15, 10, 1966), X, date(X, GrandMere, date(Y, sarah, 101)), ...

Nous pouvons déterminer comment les instructions sont formées dans le langage de la logique des prédicats.

2.2.5 Définition (formules)

Soit (Σ, Δ) une signature et $\mathcal{V}$ un ensemble de variables. L'ensemble des formules atomiques sur (Σ, Δ) et $\mathcal{V}$ est défini par $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) = \{p(t_1, \cdots, t_n) \mid p \in \Delta_n$ pour certains n, $t_1, \cdots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$.
$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ désigne l'ensemble de toutes les formules sur (Σ, Δ) et $\mathcal{V}$. Ici $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ est le plus petit ensemble avec LLOYD, 1984 :

— $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
— Si $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, puis $\neg\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
— Si $\varphi_1, \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, alors $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \leftrightarrow \varphi_2) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
— Si X $\in \mathcal{V}$ et $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, alors $(\forall X\, \varphi)$, $(\exists X\, \varphi) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$.

Pour une formule φ, $\mathcal{V}(\varphi)$ désigne l'ensemble de toutes les variables de φ. Une variable X est libre dans une formule φ ssi :

— φ est une formule atomique et X $\in \mathcal{V}(\varphi)$ ou
— $\varphi = \neg\varphi_1$ et X est libre dans φ_1 ou
— $\varphi = (\varphi_1 \cdot \varphi_2)$ avec $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ et X est libre en φ_1 ou en φ_2 ou
— $\varphi = (QY\, \varphi_1)$ avec $Q \in \{\forall, \exists\}$, X est libre en φ_1 et X$\neq$Y.
— Une formule φ est appelée fermée ssi : il n'y a pas de variable libre dans φ. Une formule est dite sans quantificateur si elle ne contient pas les caractères $\forall$ ou $\exists$.

$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ est un langage formel dans lequel des faits non seulement mathématiques, mais aussi des faits de la vie quotidienne peuvent être formulés. Dans ce qui suit, nous omettons principalement les parenthèses.

2.2.6 Exemple

Soit (Σ, Δ) à nouveau la signature de l'exemple 2.2.2. Ensuite, vous pouvez créer des formules comme suit :

$$\text{femelle(sarah)} \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \tag{2.1}$$

$$\text{mere(X, hoda)} \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \tag{2.2}$$

$$\text{ne(sarah, date(15, 10,1966))} \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \tag{2.3}$$

$$\forall F\ (\text{marie(ali, F)} \wedge \text{mere(F, K)}) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V}) \tag{2.4}$$

$$\text{marie(ali, F)} \wedge \neg(\forall F\ \text{mere (F, K)}) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V}) \tag{2.5}$$

Dans la formule de la ligne (2.2), la variable X est libre. Pour la formule φ de la ligne (2.4) $\mathcal{V}(\varphi) = \{F, K\}$ s'applique, mais seule la variable K est libre ici. Dans la formule de la ligne (2.5), cependant, F et K sont toutes les deux libres. Nous abrégeons les formules du type « $\forall X_1(...(\forall X_n\ \varphi)...)$ » souvent par « $\forall X_1, ..., X_n\ \varphi$ ». Analogue à ceci est « $\exists X_1, ..., X_n\ \varphi$ » est une abréviation de « $\exists X_1\ (...\ (\exists X_n\ \varphi)...)$ ».

2.2.7 Exemple

Le programme logique du chapitre 1 correspond à l'ensemble de formules suivant pour la signature de l'exemple 2.2.2. Comme mentionné, les variables sont toutes quantifiées dans le programme logique :
femelle(sarah).
femelle(serine).
femelle(souad).
femelle(hoda).
femelle(samia).
male(karim).
male(amine).
male(ali).
male(omar).
male(bachir).
marie(karim, sarah).
marie(ali, souad).
marie(amine, hoda).
mere(sarah, serine).
mere(sarah, amine).
mere(souad, hoda).
mere(souad, omar).
mere(hoda, samia).
mere(hoda, bachir).
$\forall$ X humain (X).
$\forall$ V, F, K marie(V, F) $\wedge$ mere(F, K) $\rightarrow$ pere(V, K).
$\forall$ X,Y mere(X, Y) $\rightarrow$ parent(X, Y).
$\forall$ X,Y pere(X, Y) $\rightarrow$ parent(X, Y).
$\forall$ V,X parent(V, X) $\rightarrow$ ancetre(V, X).
$\forall$ V,Y,X parent(V, Y) $\wedge$ ancetre(Y, X) $\rightarrow$ ancetre(V, X).
Enfin, nous avons introduit le terme substitution. Les substitutions permettent le remplacement (ou « instanciation ») de variables par des termes.

2.2.8 Définition (substitution)

Une application $\sigma : \mathcal{V} \rightarrow (\sigma, \mathcal{V})$ est appelée substitution ssi. $\sigma(X) \neq$ X n'est valable que pour un nombre fini de X $\in \mathcal{V}$. $DOM(\sigma) = \{ X \in \mathcal{V} \mid \sigma(X) \neq X\}$ est appelé le domaine de σ et RANGE(σ) = {σ (X) | X $\in$ DOM(σ)} est appelé la plage de σ. Puisque DOM(σ) est fini, il y a substitution σ comme ensemble fini de paires {X / σ (X) | X $\in$ DOM(σ)} peut être représenté. Une substitution σ est une substitution basique sur DOM(σ) ssi. $\mathcal{V}(\sigma(X)) = \emptyset$ pour tout X $\in DOM(\sigma)$. Une substitution σ est une variable renommée ssi. σ est injectif et $\sigma(X) \in \mathcal{V}$ est vrai pour tout X $\in \mathcal{V}$. Les substitutions sont des mappages de variables en termes (étendues de manière homomorphe). $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$, c'est-à-dire $\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n)$ .De la même manière, les substitutions peuvent également être étendues aux formules CHEN et TSAI, 2014 :

1. $\sigma(p(t_1, ..., t_n)) = p(\sigma(t_1), ..., \sigma(t_n))$
2. $\sigma(\neg \varphi) = \neg \sigma(\varphi)$
3. $\sigma(\varphi_1 \cdot \varphi_1) = \sigma(\varphi_1) \cdot \sigma(\varphi_2)$ pour $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
4. $\sigma(Q\ X\ \varphi) = QX\ \sigma(\varphi)$ pour $Q \in \{\forall, \exists\}$, si X $\notin \mathcal{V}(Range(\sigma)) \cup DOM(\sigma)$
5. $\sigma(QX\ \varphi) = QX'\ \sigma(\varphi)$ pour $Q \in \{\forall, \exists\}$, si X $\in \mathcal{V}(Range(\sigma)) \cup DOM(\sigma)$. Ici X' est une nouvelle variable avec $X' \notin \mathcal{V}(Range(\sigma)) \cup DOM(\sigma) \cup \mathcal{V}(\varphi)$ et $\delta = \{ X, X' \}$

La condition dans le cas 4 signifie que l'application de σ à la formule $\forall X\ \varphi$ n'est possible directement que si la variable X liée par le quantificateur, n'est pas remplacée et si cela ne remplace aucune autre variable de φ par un terme qui contient X. Sinon (dans le cas 5) X doit d'abord être renommé en une nouvelle variable X' puis la substitution σ est utilisée. Un tel changement de nom de variables quantifiées est appelé renommage des variables liées. Une instance σ(t) d'un terme t (ou une instance $\sigma(\varphi)$ d'une formule sans quantificateur φ) est appelée une instance de base si $\mathcal{V}(\sigma(t)) = \emptyset$ (ou $\mathcal{V}(\sigma(\varphi)) = \emptyset$

2.2.9 Exemple

Nous reprenons à nouveau la signature de l'exemple 2.2.2. Un exemple de substitution serait alors : $\sigma = \{X/$ date(X,Y,Z), Y/sarah, Z/date(Z,Z,Z)$\}$.
Vous obtenez :

σ (date(X, Y, Z)) = date(date(X, Y, Z), sarah, date(Z, Z, Z)).
$\sigma(\forall$ Y marie(X, Y)) = $\forall Y'$ marie(date(X, Y, Z), Y').

On écrit souvent à la place de « σ(date(X, Y, Z)) » date(X, Y, Z) [X/date(X, Y, Z), Y/sarah, Z/date(Z, Z, Z)].

2.3 Sémantique de la logique des prédicats

Notre objectif est de proposer des formules $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ représentent formellement des déclarations sur le monde. Jusqu'à présent, nous avons seulement spécifié comment les formules faites $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ peut être formé. Cependant, nous n'avons pas encore défini ce que signifient réellement ces formules. Pour ce faire, les formules doivent avoir une sémantique, c'est-à-dire un sens, attribué. Ensuite, vous pouvez également spécifier quelles formules (c'est-à-dire quelles requêtes) découlent d'un ensemble donné de formules (c'est-à-dire d'un programme logique) COLMERAUER,

1990.

Afin de définir la sémantique, on utilise des interprétations qui définissent un ensemble d'objets $\mathcal{A}$, chaque symbole de fonction (syntaxique) f une fonction α_f, tout symbole de prédicat (syntaxique) p a une relation (ou sous-ensemble) α_p et chaque variable $X \in \mathcal{V}$ un objet de $\mathcal{A}$ attribué. Une interprétation mappe chaque terme à un objet de $\mathcal{A}$ et chaque formule est vraie ou fausse par rapport à une interprétation APT, 2001.

2.4 Définition (interprétation, structure, satisfiabilité, modèle)

Pour une signature (Σ, Δ) une interprétation est un triple $I = (\mathcal{A}, \alpha, \beta)$. L'ensemble $\mathcal{A}$ est appelé le porteur de l'interprétation, où $\mathcal{A} \neq \emptyset$. De plus, α est une carte que tout symbole de fonction $f \in \Sigma_n$ une fonction $\alpha_f : \mathcal{A}^n \to \mathcal{A}$ et chaque symbole de prédicat $p \in \Delta_n$ avec $n \geq 1$ un ensemble (ou une relation) $\alpha_p \subseteq \mathcal{A}_n$ est attribué. Pour $p \in \Delta_0$, nous avons $\alpha_p \in \{TRUE, FALSE\}$. La fonction α_f et la relation α_p sont appelées interprétation du symbole de fonction f ou du symbole de prédicat p sous l'interprétation I. L'application $\beta : \mathcal{V} \to \mathcal{A}$ est appelée affectation de variable pour l'interprétation I. Pour toute interprétation I on obtient une fonction $I : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{A}$ comme suit LLOYD, 1984 :

$$I(X) = \beta(X) \text{ pour tout } X \in \mathcal{V}$$
$$I(f(t_1, ..., t_n)) = \alpha_f(I(t_1), ..., I(t_n)) \text{ pour tout } f \in \Sigma_n \text{ et } t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$$

I(t) s'appelle l'interprétation du terme t sous l'interprétation I.
Pour $X \in \mathcal{V}$ et $a \in \alpha$, $\beta[\![X/a]\!]$ est l'occupation variable avec $\beta[\![X/a]\!](X) = a$ et $\beta[\![X/a]\!](Y) = \beta(Y)$ pour tout $Y \in \mathcal{V}$ avec $Y \neq X$. $I[\![X/a]\!]$ dénote l'interprétation $(\mathcal{A}, \alpha, \beta[\![X/a]\!])$.
Une interprétation $I = (\mathcal{A}, \alpha, \beta)$ satisfait une formule $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, écrite « $I \models \varphi$ », ssi.
$\varphi = p(t_1, ..., t_n)$ et $p \in \Delta_n$ avec $(I(t_1), ..., I(t_n)) \in \alpha_p$ pour $n \geq 1$
ou $\varphi = p$ et $p \in \Delta_n$ avec α_p = TRUE
ou $\varphi = \neg\varphi_1$ et $I \not\models \varphi_1$
ou $\varphi = \varphi_1 \wedge \varphi_2$ et $I \models \varphi_1$ et $I \models \varphi_2$
ou $\varphi = \varphi_1 \vee \varphi_2$ et $I \models \varphi_1$ ou $I \models \varphi_2$
ou $\varphi = \varphi_1 \to \varphi_2$ et si $I \models \varphi_1$, alors aussi $I \models \varphi_2$
ou $\varphi = \varphi_1 \leftrightarrow \varphi_2$ et $I \models \varphi_1$ ssi. $I \models \varphi_2$
ou $\varphi = \forall X\, \varphi_1$ et $I[\![X/a]\!] \models \varphi_1$ pour tout $a \in \mathcal{A}$
ou $\varphi = \exists X\, \varphi_1$ et $I[\![X/a]\!] \models \varphi_1$ pour certains $a \in \mathcal{A}$

Une interprétation I est appelée un modèle de φ. $I \models \varphi$. I est un modèle d'un ensemble de formules Φ(« $I \models \Phi$ ») ssi. $I \models \varphi$ pour tout $\varphi \in \Phi$ est vrai. Deux formules φ_1 et φ_2 sont dites équivalentes pour toutes les interprétations I, s'applique : $I \models \varphi_1$ ssi. $I \models \varphi_2$.

Une formule ou un ensemble de formules est dit satisfiable si elle a un modèle, et insatisfaisable si elle n'a pas de modèle. Il est appelé universel si chaque interprétation est un modèle.

Une interprétation sans affectation de variable $S = (\mathcal{A}, \alpha)$ est appelée une structure. Une structure a donc un ensemble de porteuses, où les symboles de fonction de Σ étant affectés à des fonctions sur cet ensemble de porteuses et les symboles de prédicat de relations Δ étant affectés à l'ensemble de porteuses.

Si l'on ne considère que des formules fermées, on peut donc définir la satisfiabilité et le modèle à l'aide de structures. Une structure S satisfait alors une formule fermée φ (c'est-à-dire « S $\models \varphi$ » ou S est un modèle de φ) ssi. I $\models \varphi$ pour une interprétation de la forme I = ($\mathcal{A}$, α, β) est valable. La variable d'occupation β ne joue ici aucun rôle, puisque φ ne contient aucune variable libre. De même, on peut déjà définir S(t) pour les termes de base t (l'interprétation du terme t sous la structure S).

2.4.1 Exemple

Nous considérons toujours la signature de l'exemple 2.2.2. Une interprétation de cette signature est, par exemple, I = ($\mathcal{A}$, α, β) avec :

$\mathcal{A}$ = $\mathbb{N}$
α_n = n pour tout n $\in \mathbb{N}$
α_{sarah} = 0
α_{serine} = 1
α_{souad} = 2
$\vdots$
$\alpha_{date}(n_1, n_2, n_3) = n_1 + n_2 + n_3$ pour tout $n_1, n_2, n_3 \in \mathbb{N}$
$\alpha_{femelle}$ = $\{n \mid n$ est pair $\}$
α_{male} = $\{n \mid n$ est impair $\}$
α_{humain} = $\mathbb{N}$
α_{marie} = $\{(n, m) \mid n > m\}$

$\vdots$
$\beta(X)$ = 0
$\beta(Y)$ = 1
$\beta(Z)$ = 2
$\vdots$

Alors I (date(1, X, serine)) = $\alpha_{date}(alpha_1, \beta(X), \alpha_{serine})$ = 1 + 0 + 1 = 2. Ainsi, par ex.

$$\text{I} \models \text{marie(date(1, X, serine), serine).}$$

La même chose s'applique

$$\text{I} \models \forall \text{ X femelle(date(X, X, sarah)),}$$

car pour tout a $\in \mathcal{A}$ I[[X/a]] (date(X, X, sarah)) = a + a + 0 est un nombre pair. Puisque la formule $\forall$ X est féminine (date(X, X, sarah)), la structure S = ($\mathcal{A}, \alpha$) est déjà un modèle de la formule, i.e.

$$\text{S} \models \forall \text{ X femelle(date(X, X, sarah)).}$$

Les formules ci-dessus peuvent toutes être remplies, mais elles ne sont généralement pas valables, car elles ne sont pas remplies par chaque interprétation. Des exemples de formules générales sont $\varphi \vee \neg\varphi$ pour toutes les formules φ.
Des exemples de formules insatisfaisables sont $\varphi \wedge \neg\varphi$ pour toutes les formules φ.
Le lien entre le terme syntaxique « substitution » et le terme sémantique « allocation variable » est décrit par le lemme suivant.

2.4.2 Lemme (Substitution)

Soit I = ($\mathcal{A}$, α, β) une interprétation d'une signature (Σ, Δ), soit $\sigma = \{X_1/t_1, ..., X_n/t_n\}$ une substitution. Ensuite APT, 2001 :

(a) I(σ(t)) = $I[\![X_1/\ \mathrm{I}\ (t_1),\ldots, X_n/\ \mathrm{I}\ (t_n)]\!]$(t) pour tout t $\in \mathcal{T}(\Sigma, \mathcal{V})$

(b) I $\models \sigma(\varphi)$ ssi. $I[\![X_1/\ \mathrm{I}\ (t_1),\ldots, X_n/\ \mathrm{I}\ (t_n)]\!] \models \varphi$ pour tout $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

Preuve.

(a) La preuve est effectuée par induction structurelle sur la structure du terme t. Au début de l'induction, t est soit une variable, soit une constante (c'est-à-dire un symbole de fonction de Σ_0).
Si t est une variable X_i, alors

$$\mathrm{I}\ (\sigma(X_i)) = \mathrm{I}\ (t_i) = \mathrm{I}\ [\![X_1/\ \mathrm{I}\ (t_1),\ldots, X_n/\mathrm{I}(t_n)]\!]\ (X_i).$$

Si t est une variable Y différente de tout $X_1,\ldots,X_n$, on obtient

$$\mathrm{I}\ (\sigma(Y)) = \mathrm{I}\ (\mathrm{Y}) = \beta\ (\mathrm{Y}) = \mathrm{I}[\![X_1/\mathrm{I}\ (t_1),\ldots, X_n/\mathrm{I}(t_n)]\!]\ (\mathrm{Y}).$$

Si t a la forme $f\ (s_1, ..., s_k)$ (où k = 0 est possible), alors I (σ(t)) = I(σ(f(s_1,..., s_k))) = α_f(I($\sigma(s_1)$),...,I ($\sigma(s_k)$)).

L'hypothèse d'induction implique

$$\mathrm{I}(\sigma(s_i)) = \mathrm{I}\ [\![X_1/\mathrm{I}(t_1), \ldots, X_n/\mathrm{I}(t_n)]\!](s_i).$$

On obtient, donc :
α_f(I($\sigma(s_1)$),...,I($\sigma(s_k)$))
= α_f($I[\![X_1/\mathrm{I}(t_1),...,X_n/\mathrm{I}(t_n)]\!](s_1)$,...,I$[\![X_1/\mathrm{I}(t_1),...,X_n/\mathrm{I}(t_n)]\!](s_k)$)
= I$[\![X_1/\mathrm{I}(t_1),...,X_n/\mathrm{I}(t_n)]\!](f(s_1,...,s_k))$
= I$[\![X_1/\mathrm{I}(t_1),\ldots,X_n/\mathrm{I}\ (t_n)]\!]$ (t)

(b) Nous utilisons une induction structurelle sur la structure de formule φ. Si φ = $p(s_1,\ldots,s_m)$, alors s'applique
I $\models \sigma(\varphi)$
ssi.I$\models p(\sigma(s_1), \ldots, \sigma(s_m))$
ssi.(I($\sigma(s_1)$,...,I($\sigma(s_m)$)) $\in \alpha_p$
ssi.(I$[\![X_1\ /\ \mathrm{I}(s_1), ..., X_n\ /\ \mathrm{I}(s_m)]\!]\ (s_1)$,...,I$[\![X_1\ /\ \mathrm{I}(s_1), ..., X_n\ /\ I(s_m)]\!](s_m) \in \alpha_p$

selon la partie (a). Alors vous obtenez :
(I$[\![X_1 = \mathrm{I}\ (s_1), \ldots, X_n = \mathrm{I}\ (s_m)]\!](s_1)$,...,I $[\![X_1 = \mathrm{I}\ (s_1), \ldots, X_n = \mathrm{I}\ (s_m)]\!](s_m)) \in \alpha_p$
ssi. I$[\![X_1\ /\ \mathrm{I}\ (s_1), ..., X_n\ /\ \mathrm{I}\ (s_m)]\!] \models p(s_1, ..., s_m)$
ssi. I$[\![X_1\ /\ \mathrm{I}\ (s_1), ..., X_n\ /\ \mathrm{I}\ (s_m)]\!] \models \varphi$
Les cas $\varphi = \neg\varphi_1$ ou $\varphi = \varphi_1 \cdot \varphi_2$ avec $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ sont très simples. Considérons le cas $\varphi = \forall$ X φ_1 (le cas $\varphi = \exists$ X φ_1 est analogue). S'il y a. on ne considère que le cas X $\notin \mathcal{V}$(RANGE(σ))$\cup$DOM (σ)(sinon la formule est d'abord convertie en $\forall X' \varphi_1[X/X']$, ce qui équivaut évidemment à φ.) On a $\sigma(\varphi) = \forall$ X $\sigma(\varphi_1)$ et

$$\mathrm{I} \models \sigma(\varphi) \text{ ssi. I } [\![\ \mathrm{X}\ /\ \mathrm{a}\ ]\!] \models \sigma(\varphi_1) \text{ pour tout a} \in \mathcal{A}.$$

Soit $\sigma = \{X_1/t_1,\ldots, X_n/t_n\}$ (où, selon l'hypothèse, X n'ai aucune des variables X_i). Après l'hypothèse d'induction, cela se pose

$$\mathrm{I}\ [\![\ \mathrm{X}\ /\ \mathrm{a}\ ]\!] \models \sigma(\varphi_1) \text{ pour tout a} \in \mathcal{A}.$$
$$\text{ssi. I } [\![\mathrm{X/a}]\!][\![X_1/\mathrm{I}\ (t_1),\ldots,X_n/\mathrm{I}(t_n)]\!] \models \varphi_1 \text{ pour tout a} \in \mathcal{A}.$$

Puisque X est différent de tout $X_1,\ldots,X_n$, puis I $[\![$X / a $]\!][\![X_1/$ I (t_1) $,\ldots,X_n$ / I $(t_n)]\!]$ = I $[\![X_1$ / I $(t_1),\ldots,$ $X_n/$I$(t_n)]\!][\![$ X / a$]\!]$ et on obtient ainsi :

$$\begin{array}{lll} & \text{I } [\![X_1 \text{ / I } (t_1),\ldots, X_n/\text{I}(t_n)]\!][\![\text{ X / a}]\!] & \models \varphi_1 \text{ pour tout a} \in \mathcal{A} \\ \text{ssi.} & \text{I } [\![X_1/\text{I } (t_1),\ldots, X_n/\text{I } t_n)]\!] & \models \forall \text{ X } \varphi_1 \\ \text{ssi.} & \text{I } [\![X_1/\text{I } (t_1),\ldots, X_n/\text{I } t_n)]\!] & \models \varphi. \end{array}$$

2.4.3 Exemple

Nous considérons l'interprétation I de l'Exemple 2.2.2, la substitution σ = { X/date(1, X, serine)} et le terme t = date(X, Y, Z). Puis s'applique

$$\begin{array}{rl} \text{I } (\sigma(\text{t})) & = \text{I (date(date(1, X, serine), Y, Z))} \\ & = \alpha_1 + \beta(\text{X}) + \alpha_{serine} + \beta \text{ (Y)} + \beta \text{ (Z)} \\ & = 1 + 0 + 1 + 1 + 2 \\ & = 5. \end{array}$$

La même chose s'applique :

$$\begin{array}{rl} \text{I } [\![\text{ X/I (date(1, X, serine))}]\!] \text{ (t)} & = \text{I } [\![\text{X/2}]\!] \text{ (date(X, Y, Z))} \\ & = 2 + \beta \text{ (Y)} + \beta \text{ (Z)} \\ & = 2 + 1 + 2 \\ & = 5. \end{array}$$

Nous définissons maintenant quand une autre formule découle d'un ensemble de formules. C'est exactement la question qui est examinée lors de l'exécution de programmes logiques.

2.4.4 Définition

D'un ensemble de formules Φ suit la formule φ (abrégée« $\Phi \models \varphi$ ») ssi. pour toutes les interprétations I avec I $\models \Phi$ on a I $\models \varphi$. Si Φ et φ ne contiennent aucune variable libre, cela équivaut à l'exigence que pour toutes les structures S avec $\mathcal{S} \models \Phi$ nous avons $\mathcal{S} \models \varphi$. (Le signe « $\models$ » représente à la fois par une interprétation ainsi que par inférabilité à partir d'un ensemble de formules. Ce que l'on entend dans chaque cas peut être reconnu par s'il existe une interprétation ou un ensemble de formules à gauche du signe « $\models$ » .) Au lieu de « $\Phi \models \varphi$ », on écrit généralement « $\models \varphi$ » (c'est-à-dire que la formule φ est généralement valide) COLMERAUER, 1990.

2.4.5 Exemple

Soit Φ l'ensemble des formules de l'exemple 2.2.7 qui correspond au programme logique du chapitre 1. Maintenant, si vous avez une requête comme

?- male(ali).

Cela signifie que l'on essaie de prouver $\Phi \models$ male(ali). Ceci est bien sûr vrai dans l'exemple ci-dessus, puisque la formule male(ali) est contenue dans Φ. La même chose s'applique également $\Phi \models$ humain(ali), puisque Φ contient la formule $\forall$X humain(X). La requête

?- mere(X, hoda).

signifie que l'on veut rechercher si $\Phi \models \exists$ X mere(X, hoda) tient. Comme mentionné, les variables X sont toutes quantifiées dans le programme logique et quantifiées d'existence dans les requêtes. Puisque Φ contient la formule mere(souad, hoda),

nous avons $\Phi \models \exists$ X mere(X, hoda). La variable X doit être affectée de la même manière que l'interprétation de souad. Pour l'exécution de programmes logiques, il faut trouver un ensemble de formules Φ (le programme logique) et une formule Φ (la requête) si $\Phi \models \varphi$ est valable.

Dans le chapitre suivant, nous montrerons comment cette question peut être examinée automatiquement.

Chapitre 3

Résolution

3.1 Introduction

Le concept d'« inférabilité » LUGER et STUBBLEFIELD, 2008 est défini de manière sémantique. Cependant, cette définition ne convient pas pour une vérification assistée par ordinateur, car pour étudier $\models \varphi$, nous avons dû considérer toutes les interprétations (infiniment nombreuses) jusqu'à présent. Pour chaque interprétation, il fallait savoir s'il s'agit d'un modèle de Φ et dans ce cas vérifier s'il s'agit également d'un modèle de φ.

Afin d'étudier la conclusion d'une manière syntaxique (et automatisable) à la place, un calcul est utilisé. Un calcul se compose de certaines règles (principalement purement syntaxiques) qui nous permettent de dériver de nouvelles formules à partir d'un ensemble de formules. La vérification si φ de Φ peut alors être dérivé de manière syntaxique respectivement.

Pour qu'un tel calcul vérifie l'inférabilité, il doit bien sûr être correct, c'est-à-dire que si peut être dérivé de Φ, alors l'inférabilité $\Phi \models \varphi$ doit être valable. Si le pas inverse est également valide, c'est-à-dire si $\models \varphi$ implique également que φ peut être dérivé de Φ, alors le calcul est dit complet.

Dans ce chapitre, nous introduisons le calculateur de résolution, qui est utilisé en programmation logique pour étudier $\Phi \models \varphi$. Nous montrons que ce calcul est à la fois correct et complet, c'est-à-dire que dans ce cas, inférabilité et déductibilité correspondent.

L'idée du calcul de résolution est de réduire le problème d'inférence $\Phi \models \varphi$ à un problème d'insatisfiabilité, puis d'étudier ce problème d'insatisfiabilité par résolution. Le lemme suivant montre comment chaque problème d'inférabilité peut être converti en un problème d'insatisfiabilité.

3.1.1 Lemme (Conversion de problèmes d'inférabilité en problèmes d'insatisfiabilité)

Soit $\varphi_1, \ldots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$. Alors $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ LLOYD, 1984 ssi. La formule $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ n'est pas satisfiable.

Preuve.

Nous avons	$\{\varphi_1, \ldots, \varphi_k\} \models \varphi$
ssi.	pour toutes les interprétations I avec I $\models \{\varphi_1, \ldots, \varphi_k\}$ on a $\models \varphi$
ssi.	il n'y a pas d'interprétation I avec I $\models \{\varphi_1, \ldots, \varphi_k\}$ et I $\models \neg\varphi$
ssi.	$\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ est insatisfiable

3.1.2 Exemple

Montrer que dans le programme logique avec le fait :
mere(souad, hoda).
Le but :

?- mere(X, hoda).

est prouvable, il faut montrer que :

$$\{\text{mere(souad, hoda)}\} \models \exists \text{ X mere(X, hoda)}$$

s'applique. Au lieu de cela, on peut maintenant prouver l'insatisfiabilité de la formule suivante :

$$\text{mere(souad, hoda)} \wedge \neg \exists \text{ X mere(X, hoda)}$$

Le problème de la preuve de l'insatisfiabilité d'un ensemble de formules (et donc aussi le problème de la recherche de leur inférabilité) est généralement indécidable. Il n'y a donc pas de processus automatique qui se termine toujours et peut vérifier l'insatisfiabilité de chaque formule. Cependant, l'insatisfiabilité (et donc aussi l'inférence) est semi-décidable. Il existe donc un algorithme qui découvre l'insatisfiabilité de chaque formule insatisfiable en un temps fini, mais qui ne peut pas la terminer avec des formules satisfiables. Pour le problème d'inférence, cela signifie que l'on peut toujours prouver $\Phi \models \varphi$ avec l'algorithme en temps fini, s'il est vrai. Mais si $\Phi \models \neg\varphi$ est vrai, l'algorithme peut ne pas se terminer. Puisque le calcul de résolution est une procédure correcte et complète pour vérifier l'insatisfabilité d'une formule, il s'agit d'un tel processus de semi-décision. Si la formule ne peut pas être remplie, cela peut également être prouvé en automatisant le calcul de la résolution. Cependant, s'il peut être satisfait, l'automatisation du calcul de la résolution peut ne pas se terminer.

Nous avons introduit le principe de résolution pour vérifier l'insatisfiabilité d'une formule φ en quatre étapes. L'idée de base ici est de réduire le principe de résolution pour la logique des prédicats au principe de résolution en logique propositionnelle. En logique propositionnelle, l'exactitude et l'exhaustivité de la résolution sont plus faciles à prouver BRATKO, 2011. (Dans la logique propositionnelle, la résolution est également une expérience de décision pour l'insatisfiabilité.)

1. Nous montrons d'abord que φ peut être converti en forme normale de Skolem pour vérifier l'insatisfiabilité. La forme normale peut être convertie (section 3.2). Les formules sous forme normale de Skolem sont fermées, elles n'ont plus de quantificateurs existentiels et elles ne contiennent que des quantificateurs universels à l'extérieur. Ces formules ont la forme $\forall X_1, \ldots, X_n \psi$, où ψ est sans quantificateur et aucune variable ne contient sauf $X_1, \ldots, X_n$.
2. Ensuite, nous montrons que dans le cas des formules sous forme normale de Skolem, il n'est pas nécessaire de considérer toutes les interprétations pour étudier l'insatisfiabilité.
3. Afin d'obtenir une procédure plus efficace, nous étendons finalement la résolution propositionnelle à la logique des prédicats en utilisant la procédure d'unification.

3.2 Forme normale de Skolem

Le premier objectif est de convertir des formules en une forme normale de la forme $\forall X_1, \ldots, X_n\ \psi$, où ψ est sans quantificateur et aucune variable ne contient sauf $X_1, \ldots, X_n$. C'est fait en deux étapes NILSSON et MALUSZYNSKI, 1995. Tout d'abord, la formule est convertie en forme normale Prenexe.

3.2.1 Définition (Forme Normale Prenexe)

Une formule φ est sous forme normale prenexe ssi. elle est de la forme suivante : $\mathcal{Q}_1\ X_1, ..., \mathcal{Q}_n\ X_n\ \psi$ avec $\mathcal{Q}_i \in \{\forall, \exists\}$ où ψ est sans quantificateur.

Le théorème suivant montre que (et comment) convertir une formule en une formule équivalente en une forme normale de prenexe NILSSON et MALUSZYNSKI, 1995.

3.2.2 Théorème (Conversion en forme normale de prenexe)

Pour toute formule φ une formule $\varphi^{'}$ en forme normale de prenexe peut être automatiquement construite, de sorte que φ et $\varphi^{'}$ soient équivalents ROWE, 1988.

Preuve.

D'abord, toutes les sous-formules $\varphi_1 \leftrightarrow \varphi_2$ dans φ sont remplacées par $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$. Et toutes les sous-formules $(\varphi_1 \rightarrow \varphi_2)$ dans φ sont remplacées par $\neg\ \varphi_1 \vee \varphi_2$. La formule restante est convertie à l'aide de l'algorithme PRENEXE suivant, dont le timing et l'exactitude sont évidents. L'algorithme reçoit n'importe quelle formule φ en entrée sans les jonctions « $\leftrightarrow$ » et « $\rightarrow$ » et délivre une formule équivalente sous forme normale Prenexe en sortie.

— Si φ est sans quantificateur, alors renvoie φ.
— Si $\varphi = \neg\ \varphi_1$, alors calculez PRENEXE $(\varphi_1) = \neg\ \mathcal{Q}_1\ X_1 \ldots \neg\ \mathcal{Q}_n\ X_n \psi$. Donne au retour $\mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n \neg\ \psi$, où $\neg \forall = \exists$ et $\neg \exists \neg\ = \forall$.
— Si $\varphi = (\varphi_1 \cdot \varphi_2)$ avec $\cdot \in \{\wedge, \vee\}$, alors calculez PRENEXE $(\varphi_1) = \mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n\ \psi_1$ et PRENEXE $(\varphi_2) = \mathcal{R}_1\ Y_1 \ldots \mathcal{R}_m\ Y_m\ \psi_2$. En renommant les variables liées, nous obtenons $X_1, \ldots, X_n$ pas dans $\mathcal{R}_1\ Y_1 \ldots \mathcal{R}_m\ Y_m\ \psi_2$ se produisent et que $Y_1, \ldots, Y_m$ pas dans $\mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n\ \psi_1$ se produisent. Maintenant, retournez la formule suivante :
$$\mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n \mathcal{R}_1\ Y_1 \ldots \mathcal{R}_m\ Y_m\ (\psi_1 \cdot \psi_2)$$
— Si $\varphi = \mathcal{Q}X\varphi_1$ avec $\mathcal{Q} \in \{\forall, \exists\}$, alors calculer la formule PRENEXE $(\varphi_1) = \mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n\ \psi$. En renommant les variables liées, nous y parvenons $X_1, \ldots, X_n$ sont différents de X. Puis donne au retour $\mathcal{Q}X\mathcal{Q}_1\ X_1 \ldots \mathcal{Q}_n\ X_n\ \psi$.

3.2.3 Exemple

Nous considérons la conversion de la formule suivante :

$$\neg\ \exists\ \text{X (marie(X, Y)} \vee \neg\ \exists\ \text{Y mere(X, Y))}$$

Nous calculons d'abord PRENEXE ($\neg\ \exists$ Y (mere(X, Y)) = $\forall\ \neg$ Y mere(X, Y). Pour que la variable liée Y dans cette sous-formule soit différente de la variable libre Y dans la sous-formule marie(X, Y), nous renommons la première en Z et obtenons ainsi $\forall\ \neg$ mere (X, Z).

Maintenant, nous calculons PRENEXE (marie(X, Y) $\vee \neg \exists$ Y mere(X, Y)), ce qui donne la formule suivante :

$$\forall \text{ Z (marie(X, Y)} \vee \neg \text{ mere(X, Y))}$$

Finalement, nous obtenons PRENEXE ($\neg \exists$ X (marie(X, Y) $\vee \neg$ mere(X, Y))) = $\forall$ X $\exists$ Z $\neg$ (marie(X, Y) $\vee \neg$ mere(X, Z)).

3.2.4 Exemple

Comme un autre exemple, nous considérons la conversion de la formule mere(souad, hoda)$\wedge \neg \exists$ X mere(X, hoda) de l'Ex.3.1.2, dont l'insatisfiabilité doit être prouvée afin de montrer que la requête

?- mere(X, hoda).

dans le programme logique avec le fait

mere(souad, hoda)

est démontrable. Les résultats suivants de la conversion dans la forme normale Prenexe :

$$\forall \text{ X mere(souad, hoda)} \wedge \neg \text{ mere(X, hoda)}$$

Nous définissons maintenant la forme normale de Skolem.

3.2.5 Définition (forme normale de Skolem)

Une formule φ est sous la forme normale de Skolem ssi. il est fermé et il a la forme $\forall X_1 \ldots X_n \, \psi$, où ψ est sans quantificateur APT, 2001.

Pour convertir une formule en forme normale Skolem, la formule est d'abord convertie en forme normale Prenexe. Le transfert restant sert à éliminer les variables libres et les quantificateurs existentiels. Contrairement à la forme normale de Prenexe, il n'y a pas de formule équivalente dans la forme normale de Skolem pour chaque formule, car il y a évidemment par ex. pas de formule sous forme normale de Skolem qui équivaut au féminin (X) ou au $\exists$ X femelle (X). Cependant, pour chaque formule, il existe une formule Forme normale de Skolem.

L'idée du transfert est de créer d'abord la forme normale de Prenexe, puis de lier toutes les variables libres avec des quantificateurs existentiels, et enfin de supprimer toutes les variables quantifiées existentielles à l'aide de nouveaux symboles de prédicat.

3.2.6 Théorème (Conversion en forme normale de Skolem)

Pour toute formule φ une formule φ' sous forme normale de Skolem peut être construite automatiquement, de sorte que φ puisse être satisfait si et seulement si φ' est satisfiable LLOYD, 1984.

Preuve.

Premièrement, la formule φ est convertie en forme normale Prenexe en utilisant la procédure du théorème 3.2.2. Soit $X_1, \ldots, X_n$ sont les variables libres dans la formule résultante φ_1 qui conduisent à φ est équivalent. Ensuite, φ_1 est converti davantage

en la formule fermée φ_2 de la forme $\exists X_1, \ldots, X_n\ \varphi_1$, ce qui équivaut à la satisfiabilité φ_1 : De I $\models\ \varphi_1$ suit I $[\![X_1/\beta(X_1), \ldots, X_n/\beta(X_n)]\!]\ \models\ \varphi_1$ et donc évidemment aussi I $\models\ \exists X_1, \ldots, X_n\ \varphi_1$. Inversement, il résulte de I $\models\ \exists X_1, \ldots, X_n\ \varphi_1$, qu'il y a $a_1, \ldots, a_n$ indique tel que I $[\![X_1/a_1, \ldots, X_n/a_n]\!]\ \models\ \varphi_1$ est vrai.
La formule φ_2 est donc fermée, sous forme normale Prenexe et équivalente à la satisfiabilité φ. Maintenant, nous éliminons les quantificateurs existentiels étape par étape de l'extérieur vers l'intérieur BRATKO, 2011. Si φ_2 a la formule $\forall X_1, \ldots, X_n\ \exists Y\ \psi$. φ_2 par la formule suivante :

$$\forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)].$$

Toutes les occurrences de Y sont remplacées par $f(X_1, \ldots, X_n)$, où f est une nouvelle est un symbole de fonction à n arguments. Cette procédure est répétée jusqu'à ce qu'il n'y ait plus de quantificateurs existentiels. La formule résultante est la satisfiabilité équivalente à φ_2 et donc aussi à φ. Ce qui suit s'applique :

I $\models\ \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$.
$\curvearrowright$ I $[\![X_1/a_1, \ldots, X_n/a_n]\!]\ \models\ \psi[Y/f(X_1, \ldots, X_n)]$ pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$
$\curvearrowright$ I $[\![X_1/a_1, \ldots, X_n/a_n]\!]\ [Y/$ I $[\![X_1/a_1, \ldots, X_n/a_n]\!](f(X_1, \ldots, X_n)]\ \models\ \psi$
pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$, grace à la lemme de substitution
$\curvearrowright$ I $[\![X_1/a_1, \ldots, X_n/a_n]\!]\ \models\ \exists Y\ \psi$ pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$
$\curvearrowright$ I $\models\ \forall X_1, \ldots, X_n\ \exists Y\ \psi$ pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$

À partir de I $\models\ \forall X_1, \ldots, X_n\ \exists Y\ \psi$ avec I = $(\mathcal{A}, \alpha, \beta)$ il s'ensuit que pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$, $b \in \mathcal{A}$ tel que I $[\![X_1/a_1, \ldots, X_n/a_n, Y/b]\!]\ \models\ \psi$. Soit F la fonction de $\mathcal{A}_n\ \rightarrow \mathcal{A}$, qui affecte le b correspondant à chaque tuple $(a_1, ..., a_n)$. Soit I$'$ = $(\mathcal{A}, \alpha', \beta')$, où α' ne diffère de α que dans l'interprétation du nouveau symbole de fonction f. Ici on définit $\alpha'_f = F$. On obtient alors I$'\ \models\ \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$. La raison est-ce pour tous $a_1, \ldots, a_n \in\ \mathcal{A}$:

I $[\![X_1/a_1, \ldots, X_n/a_n, Y/F(a_1, \ldots, a_n)]\!]\ \models\ \psi$
ssi. I$'[\![X_1/a_1, \ldots, X_n/a_n, Y/F(a_1, \ldots, a_n)]\!]\ \models\ \psi$ puisque f n'apparaît pas dans ψ
ssi. I$'[\![X_1/a_1, \ldots, X_n/a_n]\!]\ [\![Y/$ I$'[\![X_1/a_1, \ldots, X_n/a_n]\!]f(X_1, \ldots, X_n)]\!]\ \models\ \psi$
ssi. I$'[\![X_1/a_1, \ldots, X_n/a_n]\!]\ \models\ \psi[Y/f(X_1, \ldots, X_n)]$

Selon le lemme de substitution, puisque cette déclaration s'applique à tous les $a_1, \ldots, a_n \in$ $\mathcal{A}$ est vrai, est obtenu I$'\ \models\ \forall X_1, \ldots, X_n\ \psi\ [Y/f(X_1, \ldots, X_n)]$

3.2.7 Exemple

Nous considérons la transformation de la formule

$$\neg\ \exists\ \text{X (marie(X, Y)}\ \vee \neg\ \exists\ \text{Y mere(X, Y))}$$

de l'exemple sous la forme normale de Skolem. Comme dans l'exemple, la formule est d'abord transformée en forme normale Prenexe, ce qui conduit à cette formule

$$\forall\ \text{X}\ \exists\ \text{Z}\ \neg\ \text{(marie(X, Y)}\ \vee\ \exists\ \text{mere(X, Z))}$$

Alors la variable libre Y est quantifiée existentiellement, ce que résulte :

$$\exists\ \text{Y}\ \forall\ \text{X}\ \exists\ \text{Z}\ \neg\ \text{(marie(X, Y)}\vee\ \neg\ \text{mere(X, Z))}$$

Afin d'éliminer les quantificateurs existentiels, la variable quantifiée existentielle la plus externe Y est d'abord remplacée par une nouvelle constante a. Ici a est nul car il n'y a plus de quantificateurs universels avant le quantificateur existentiel de Y. Cela donne

$$\forall \, \mathrm{X} \, \exists \, \mathrm{Z} \, \neg \, (\mathrm{marie(X,\, a)} \vee \neg \, \mathrm{mere(X,\, Z)}).$$

Enfin, Z est remplacé par f(X), où f est un nouveau symbole de fonction à un argument. Cela mène à

$$\forall \, \mathrm{X} \, \neg \, (\mathrm{marie(X,\, a)} \vee \neg \, \mathrm{mere(X,\, f(X))}).$$

3.3 Résolution de base

Afin de pouvoir effectuer plus efficacement le test d'insatisfiabilité, nous avons introduit la procédure de résolution des preuves dans cette section SHOHAM, 2014. Dans cette section, nous nous limiterons à la résolution propositionnelle, puis nous l'étendrons à la résolution logique de prédicat. La résolution propositionnelle est également appelée résolution de base, car nous ne considérons que des formules sans variable (c'est-à-dire des formules avec des termes de base). Pour utiliser une formule de la forme $\forall X_1, \ldots, X_n \, \psi$ sous forme normale Skolem avec la résolution de base.

Pour examiner l'insatisfiabilité, la formule ψ doit d'abord être convertie en forme normale conjonctive (FNC). Ces formules sont alors représentées comme des ensembles de clauses.

3.3.1 Définition (Forme Normale Conjonctive, littéral, clause)

Une formule ψ est sous forme normale conjonctive (FNC) ssi. elle est sans quantificateur et a la forme suivante SHOHAM, 2014 :

$$(\mathrm{L}_{1,1} \ \vee \ldots \vee \mathrm{L}_{1,n_1}) \wedge \ldots \wedge (\mathrm{L}_{m,1} \ \vee \ldots \vee \mathrm{L}_{m,n_m}).$$

Ici les $\mathrm{L}_{i,j}$ sont des littéraux, c'est-à-dire des formules atomiques ou la négation d'une forme atomique de la forme $\mathrm{p}(t_1, \ldots, t_k)$ ou $\neg \, \mathrm{p}(t_1, \ldots, t_k)$. Nous définissons son négatif L pour un littéral $\neg$ L comme suit :

$$\neg L = \begin{cases} \neg A, & \text{si } L = A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}). \\ A, & \text{si } L = \neg A \text{ si } A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}). \end{cases}$$

Un ensemble de littéraux est appelé une clause. Chaque formule ψ dans FNC comme ci-dessus correspond à l'ensemble de clauses correspondant.

$$\mathcal{K}(\psi) = \{\{\mathrm{L}_{1,1}, \ldots, \mathrm{L}_{1,n_1}), \ \ldots \ , (\mathrm{L}_{m,1}, \ldots, \mathrm{L}_{m,n_m}\}\}.$$

Une clause représente la disjonction entièrement quantifiée de ses littéraux et un ensemble de clauses représente la conjonction de ses clauses. Dans ce qui suit, nous parlons donc également de faisabilité et d'inférence d'ensembles de clauses. Sauf indication contraire explicite, nous ne considérons que des ensembles finis de clauses dans ce qui suit. La clause vide est souvent écrite comme $\square$ et par définition elle n'est pas satisfaisante.

Une conversion en FNC (et donc en représentation sous forme d'un ensemble de clauses) est facilement possible automatiquement pour chaque formule sans quantificateur.

3.3.2 Théorème (conversion en FNC)

Pour toute formule sans quantificateur ψ, une formule $\psi^{'}$ peut être construite automatiquement sous Forme Normale Conjonctive (FNC), de sorte que ψ et $\psi^{'}$ sont équivalents CHEN et TSAI, 2014.

Preuve.
Tout d'abord, toutes les formules partielles $\psi_1 \leftrightarrow \psi_2$ dans ψ sont remplacées par$\psi_1 \rightarrow \psi_2 \wedge \psi_2 \rightarrow \psi_1$. Alors toutes les sous-formules $\psi_1 \rightarrow \psi_2$ dans ψ sont remplacées par $\neg\, \psi_1 \vee \psi_2$. La conversion de la formule restante est effectuée avec l'algorithme FNC suivant, dont le timing et l'exactitude sont évidents. L'algorithme reçoit toute formule sans quantificateur ψ en entrée sans les jonctions « $\leftrightarrow$ » et « $\rightarrow$ » et délivre une formule équivalente en FNC en sortie.

- Si ψ est atomique, alors renvoie ψ.
- Si $\psi = \psi_1 \wedge \psi_2$, alors renvoie FNC $(\psi_1) \wedge$ FNC (ψ_2).
- Si $\psi = \psi_1 \vee \psi_2$, alors calculez FNC$(\psi_1) = \bigwedge_{i\in\{1,\ldots,m_1\}} \psi_i^{'}$ et FNC $(\psi_2) = \bigwedge_{j\in\{1,\ldots,m_2\}} \psi_i^{''}$ Ici, $\psi_i^{'}$ et $\psi_i^{''}$ sont des disjonctions de littéraux. En appliquant la loi distributive, on obtient la formule $\bigwedge_{i\in\{1,\ldots,m_1\},\, j\in\{1,\ldots,m_2\}} \psi_i^{''} \vee \psi_j^{''}$ dans FNC, qui est retourné.
- Si $\psi = \neg\, \psi_1$, alors calculez FNC $(\psi_1) = \bigwedge_{i\in\{1,\ldots,m\}} (\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$. Plusieurs Application de la règle de Morgan à $\neg \bigwedge_{i\in\{1,\ldots,m\}} (\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$ donne $\bigwedge_{i\in\{1,\ldots,m\}} (\bigvee_{j\in\{1,\ldots,n_i\}} \neg L_{i,j})$. Puis procédez comme le cas précédent et tournez la loi distributive, cela donne $\bigwedge_{j_1\in\{1,\ldots,n_1\},\ldots,j_m\in\{1,\ldots,n_m\}} \neg L_{1,j_1} \vee, \ldots, \vee \neg L_{m,j_m}$.

3.3.3 Exemple

À titre d'exemple, considérons la formule suivante, où p, q, r $\in \Delta_0$.

$$\neg\, (\neg\, \mathrm{p} \wedge (\neg\, \mathrm{q} \vee \mathrm{r}\,))$$

En appliquant la règle de Morgan, on obtient

$$\mathrm{p} \vee (\,\mathrm{q} \wedge \neg\, \mathrm{r}\,))$$

Enfin, la loi distributive donne la formule souhaitée en FNC :

$$(\mathrm{p} \vee\, \mathrm{q}) \wedge (\mathrm{p} \vee \neg\, \mathrm{r}\,))$$

3.4 Résolution et unification de la logique des prédicats

L'idée mentionnée de substitutions peut être illustrée par l'exemple suivant.

3.4.1 Exemple

Nous considérons l'ensemble de clauses suivant, où p, q $\in \Delta_1$, f $\in \Sigma_1$ et a $\in \Sigma_0$

$$\{\, \{\, \mathrm{p(X)}, \neg\, \mathrm{q(X)}\,\}, \{\neg\, \mathrm{p(f(Y))}\,\}, \{\, \mathrm{q(f(a))}\,\}\,\}$$

Pour résoudre les deux premières clauses, nous utilisons maintenant la substitution $\{$ X/f(Y) $\}$. Après avoir utilisé cette substitution, les littéraux p(X)[X/f(Y)] = p(f(Y)) et $\neg$ p(f(Y))[X/f(Y)] = $\neg$ p(f(Y)), qui sont complémentaires l'une de l'autre. La substitution $\{$ X/f(Y)$\}$ est donc un unificateur de $\}$ p(X), p(f(Y))$\}$. La résolvante est alors la clause $\{$ $\neg$ q(X)[X/f(Y)]$\}$ = $\{$ $\neg$ q(f(Y))$\}$ avec le second littéral restant de la première clause.

L'avantage est que la substitution { X/f(Y)} n'a pas encore déterminé comment Y est ensuite instancié. Il s'avère qu'il faut alors utiliser la substitution {Y/a} pour finalement dériver la clause vide. Cependant, il n'est pas nécessaire de le reconnaître d'emblée (c'est-à-dire qu'il n'est pas nécessaire d'utiliser la substitution {X/f(a)}), mais on n'utilise que des substitutions telles que {X/f(Y)}, qui sont aussi générales que possible. En d'autres termes, dans la première étape de résolution, nous choisissons l'unificateur le plus général de { p(X), p(f(Y))} et non l'unificateur moins général {X/f(a)} ROWE, 1988.

La définition suivante introduit le concept de la formule d'unification.

3.4.2 Définition (unification)

Une clause K = { L_1, ..., L_n } est unifiable ssi. il y a une substitution σ avec $\sigma(L_1)$ = ... = $\sigma(L_n)$ (c'est-à-dire | $\sigma(K)$ | = 1). Une telle substitution est appelée unificateur de K. Un unificateur σ est appelé unificateur le plus général (upg), si pour tout unificateur σ' il y a une substitution δ avec σ'(X) = δ (σ(X)) pour tout X $\in \mathcal{V}$ ROWE, 1988.

Si une clause est unifiable, il y a aussi un unificateur général qui est unique en dehors du renommage des variables. Pour une clause, on peut décider si elle est unifiable. Le premier algorithme d'unification suivant a été développé par J. Robinson LUGER et STUBBLEFIELD, 2008. En entrée, il reçoit une clause K = {L_1, ..., L_n} pour être unifié.

Algorithme d'unification

1. Soit $\sigma = \emptyset$ la substitution vide (ou «identique»).
2. Si | σ (K) | = 1, puis se termine et produit σ en upg de K.
3. Sinon, recherchez dans tous les $\sigma(L_i)$ en parallèle de gauche à droite jusqu'à ce que les caractères lus soient différents dans deux littéraux.
4. Si aucun des deux caractères n'est une variable, annulez avec échec de conflit.
5. Sinon, soit X la variable et t le terme partiel de l'autre littéral, (ici t peut aussi être une variable). Si X se produit dans t, terminez avec Échec de l'occurrence. (Celles-ci La vérification s'appelle une vérification des événements.)
6. Sinon, définissez $\sigma = \{X/t\} \circ \sigma$ et revenez à l'étape 2.

Ici $\sigma_1 \circ \sigma_2$ signifie la composition (ou « l'une derrière l'autre ») des substitutions. On a donc ($\sigma_1 \circ \sigma_2$) (X) = σ_1 (σ_2(X)).

3.4.3 Exemple

Soit q $\in \Delta_1$, p $\in \Delta_2$, f, g $\in \Sigma_2$, h $\in \Sigma_1$ et a $\in \Sigma_0$.
Comme exemple d'un échec, considérez la clause

{ q(f(X, Y)), q(g(X, Y))}.

Le premier endroit où les deux littéraux diffèrent est le symbole de fonction f et le symbole de fonction g. Si les variables X et Y ne sont pas instanciées, ces deux littéraux deviennent les mêmes, car les instanciations ne peuvent pas changer ces différents symboles de prédicat. Cette clause n'est donc pas unifiable. Comme exemple d'un échec d'occurrence, considérez la clause suivante APT, 2001 :

{ q(X), q(h(X))}.

Au premier endroit différent, il y a une fois la variable X et une fois le terme h (X), qui contient la variable X. Si X n'est pas instancié, ces termes ou les littéraux originaux peuvent être rendus identiques. La clause n'est donc pas non plus unifiable. Enfin, nous appliquons l'algorithme d'unification à la clause à partir des deux littéraux suivants LLOYD, 1984 :

$$\neg\ p(f(Z,\ g(a,Y)),\ h(Z))\}.$$
$$\neg\ p(f(\underline{f(U,\ V)},\ W),\ h(f(a,\ Y))\}.$$

Le texte souligné indique le premier point où les deux littéraux diffèrent. D'où σ = { Z/f(U, V) }. On applique maintenant l'unificateur partiel déjà trouvé aux littéraux :

$$\neg\ p(f(f(U,\ V),\ g(a,Y)),\ h(f(U,\ V)).$$
$$\neg\ p(f(f(U,\ V),\ \underline{W}),\ h(f(a,\ Y)).$$

Le résultat est σ = { W/g(a, Y)} $\circ$ σ = { Z/f(U, V), W/g(a, Y)}. En appliquant σ, on obtient :

$$\neg\ p(f(f(U,\ V),\ g(a,Y)),\ h(f(U,\ V)).$$
$$\neg\ p(f(f(U,\ V),\ g(a,Y)),\ h(f(\underline{a},\ Y)).$$

Nous obtenons maintenant σ = { U/a} $\circ$ σ = { Z/f(a, V), W/g(a, Y), U/a}. L'application de σ conduit à :

$$\neg\ p(f(f(U,\ V),\ g(a,Y)),\ h(f(U,\ V)).$$
$$\neg\ p(f(f(U,\ V),\ g(a,Y)),\ h(f(a,\ \underline{Y})).$$

Finalement, nous obtenons σ = { Y/V } $\circ$ σ = { Z/f(a, V), W/g(a, V), U/a, Y/V}. Après cela, les deux littéraux instanciés sont identiques, c'est donc le pgu que nous recherchons.

La phrase suivante montre la fin et l'exactitude de l'algorithme d'unification. Il existe une version plus formelle de l'algorithme d'unification sous la forme d'un ensemble de quatre règles de transformation, qui formalisent aussi précisément la recherche « parallèle des littéraux de gauche à droite ». Là, la terminaison et l'exactitude de cet algorithme sont prouvées avec précision.)

3.4.4 Théorème (Terminaison et correction de l'algorithme d'unification)

L'algorithme d'unification se termine pour chaque clause K et il est correct, c'est-à-dire qu'il délivre un pgu pour la clause K ssi. K est unifiable.

Preuve.

La fin de l'algorithme s'ensuit, puisque le nombre de variables dans σ (K) diminue de 1 à chaque itération de la boucle de l'étape 2 à 6. Si l'algorithme se termine avec succès et retourne une substitution σ, alors σ est évidemment un unificateur de K, puisque $|\sigma\ (K)|$ = 1 s'applique. Si la clause K est donc non unifiable, l'algorithme doit rompre avec un clash ou un échec d'occurrence.
Il reste à montrer que pour chaque clause unifiable on trouve effectivement un unificateur et que cet unificateur est alors aussi un unificateur le plus général. Soit m $\geq$ 0 le nombre d'itérations de boucle qui ont lieu lorsque la clause K est introduite. Pour tout 0 $\leq$ i $\leq$ m, soit σ_i la valeur de σ après le passage de la i^{eme} boucle. Nous montrons la revendication suivante pour tout 0 $\leq$ i $\leq$ m COLMERAUER, 1990 :

$$\text{Pour tout unificateur } \sigma' \text{ de K, nous avons } \sigma' = \sigma' \circ \sigma_i\ . \tag{3.1}$$

De l'assertion (3.1), il n'est pas possible de se terminer par un conflit ou un échec d'occurrence. Parce que si la passe de (m+1) à la i^{eme} boucle était annulée, σ_m (K) ne serait pas unifiable. Mais puisque K est unifiable, K a un unificateur $\sigma^{'} = \sigma^{'} \circ \sigma_i$. Donc $\sigma^{'}$ est aussi l'unificateur de σ_m (K).
Puisque la boucle n'est parcourue que m fois, alors $|\sigma_m$ (K)$|$ = 1, c'est-à-dire que σ_m est un unificateur de K. Puisque, de plus, pour tout unificateur $\sigma^{'}$ de K il y a une substitution $= \sigma^{'}$ avec $\sigma^{'} = \delta \circ \sigma_m$, σ_m est alors aussi pgu de K. (3.1) par récurrence sur i. Dans le début de l'induction i = 0, $\sigma_0 = \emptyset$ est l'identité. Donc $\sigma^{'} = \sigma^{'} \circ \sigma_0$ pour tout $\sigma^{'}$ est également valable. Dans l'étape d'induction i > 0, une variable X est trouvée dans un littéral et un terme t dans l'autre littéral dans la passe de la i-^{me} boucle, ce qui donne $\sigma_i = \{$ X/t $\} \circ \sigma_{i-1}$. Pour tout unificateur $\sigma^{'}$ de K, selon l'hypothèse d'induction, $\sigma^{'} = \sigma^{'} \circ \sigma_{i-1}$ s'applique. Donc il suit :

$$\begin{aligned} & \sigma^{'} \circ \sigma_i \\ = \ & \sigma^{'} \circ \{ \text{X/t} \} \circ \sigma_{i-1} \text{ (déf. de } \sigma_i) \\ = \ & \sigma^{'} \circ \sigma_{i-1} \text{ (puisque } \sigma^{'} \circ \{ \text{X/t} \} = \sigma^{'} \} \end{aligned}$$

Pour montrer $\sigma^{'} \circ \{$ X/t $\} = \sigma^{'}$ on reconnaît d'abord que les deux substitutions sont évidemment identiques sur toutes les variables Y $\neq$ X. La variable X donne ($\sigma^{'} \circ \{$ X/t $\}$) (X) = $\sigma^{'}$ (t) = $\sigma^{'}$ (X), puisque $\sigma^{'}$ est l'unificateur de σ_{i-1} (K) (car $|\sigma^{'}$ (K)$|$ = $|\sigma^{'}(\sigma_{i-1}$ (K))$|$ = 1) et tout unificateur de σ_{i-1} (K) doit également unifier les termes X et t.
Avec l'aide de l'unification, nous pouvons maintenant définir la résolution logique du prédicat.

3.4.5 Définition (Résolution logique du prédicat)

Soit les clauses K_1 et K_2. Alors la clause R est résolvante de K_1 et K_2 ssi. les trois conditions suivantes s'appliquent LUGER et STUBBLEFIELD, 2008 :

— Il existe des variables renommées v_1 et v_2, de sorte que $v_1(K_1)$ et $v_2(K_2)$ ne contiennent pas des variables communes.
— Il existe des littéraux $L_1, \ldots, L_m \in v_1(K_1)$ et les littéraux $L_1^{'}, \ldots, L_n^{'} \in v_2(K_2)$ avec n, m $\geq$ 1 tel que $\{\neg L_1, \ldots, \neg L_m, L_1^{'}, \ldots, L_n^{'}\}$ est unifiable avec un pgu σ.
— R = σ $((v_1(K_1) \backslash \{L_1, \ldots, L_m\}) \cup (v_2(K_2) \backslash \{L_1^{'}, \ldots, L_n^{'}\}))$

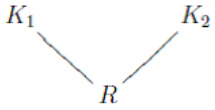
Pour un ensemble de clauses K, nous définissons la résolution suivante :

$$\begin{aligned} \text{Res}(\mathcal{K}) &= \mathcal{K} \cup \{ \text{ R } | \text{ R est la résolvante de deux clauses de } \mathcal{K}\} \\ \text{Res}^0(\mathcal{K}) &= \mathcal{K} \\ \text{Res}^{n+1}(\mathcal{K}) &= \text{Res (Res}^n(\mathcal{K})) \text{ pour tout n} \geq 0 \\ \text{Res}^*(\mathcal{K}) &= \bigcup_{n \geq 0} \text{Res}^n(\mathcal{K}) \end{aligned}$$

Evidemment, la résolution propositionnelle est un cas particulier de la résolution logique de prédicat, car dans les clauses sans variable. Analogue à la résolution propositionnelle, $\square \in \text{Res}^*(\mathcal{K})$ ssi. C'est une suite de clauses $K_1, \ldots, K_m$ telle que : $K_m = \square$ et telle que pour tout 1 $\leq$ i $\leq$ m LLOYD, 1984 :

— $K_i \in \mathcal{K}$ ou
— K_i est une résolvante de K_j et K_k pour j, k < i

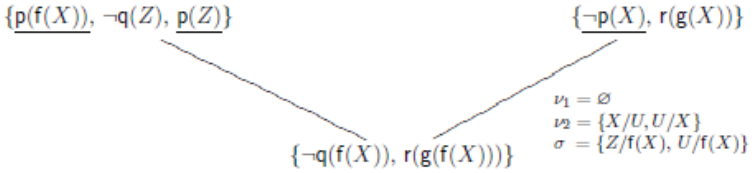
Pour illustrer les preuves de résolution, nous écrivons à nouveau le schéma suivant pour faire comprendre que R provient de la résolution de K_1 et K_2.



Une représentation plus précise est possible en soulignant les littéraux éliminés et en spécifiant explicitement le renommage et l'unificateur.

3.4.6 Exemple

À titre d'exemple, considérons le pas de résolution suivant, où p, q, r $\in \Delta_1$ et f, g $\in \Sigma_1$.



3.4.7 Lemme de la résolution de la logique des prédicats

Soit $\mathcal{K}$ un ensemble de clauses. Si $K_1, K_2 \in \mathcal{K}$ et R est résolvant de K_1 et K_2, alors $\mathcal{K}$ et $\mathcal{K} \cup \{R\}$ sont équivalents APT, 2001.

Preuve.

$S \models \mathcal{K} \cup \{R\}$ suit trivialement $S \models \mathcal{K}$ pour toutes les structures S. Il suffit de considérer des structures au lieu d'interprétations, puisque chaque clause représente la disjonction tout quantifiée de ses littéraux et chaque ensemble de clauses la conjonction de ses clauses. Ainsi, $\mathcal{K} \cup \{R\} \models \mathcal{K}$. A l'inverse, soit S une structure qui satisfait $\mathcal{K}$. On a :

$$R = \sigma\left((v_1(K_1) \backslash \{L_1, \ldots, L_m\}) \cup (v_2(K_2) \backslash \{L_1^{'}, \ldots, L_n^{'}\}\right.$$

Ici, v_1 et v_2 sont des renommages de variable, de sorte que $v_1(K_1)$ et $v_2(K_2)$ ne contiennent aucune variable commune. En outre, $L_1, \ldots, L_m \in v_1(K_1)$ et $L_1^{'}, \ldots, L_n^{'} \in v_2(K_2)$ avec n, m ≥ 1 tel que $\{\neg L_1, \ldots, \neg L_m, L_1^{'}, \ldots, L_n^{'}\}$ sont unifiables avec les pgu σ. Ça s'applique donc $\sigma(L_1) = ... = \sigma(L_m) = L$ et $\sigma(L_1^{'}) = ... = \sigma(L_n^{'}) = \neg L$ pour un littéral L. On suppose que $S \not\models \mathcal{K} \cup \{R\}$. De $S \models \mathcal{K}$ il s'ensuit que $S \not\models R$. Soit :

$$v_1(K_1) = \{L_1, \ldots, L_m, L_{m+1}, \ldots, L_p\} \text{ et } v_2(K_2) = \{L_1^{'}, \ldots, L_n^{'}, L_{n+1}^{'}, \ldots, L_q^{'}\}$$

avec p $\geq$ m et q $\geq$ n. Alors R est obtenu par quantification universelle de la formule

$$\sigma(L_{m+1} \vee ... \vee L_p \vee L_{n+1}^{'} \vee ... \vee L_q^{'})$$

Soit $S = (\mathcal{A}, \alpha)$. Il y a donc une interprétation $I = (\mathcal{A}, \alpha, \beta)$ avec :

$$I \not\models \sigma(L_{m+1} \vee ... \vee L_p \vee L_{n+1}^{'} \vee ... \vee L_q^{'})$$

Soit $\sigma = \{X_1/t_1, ..., X_k/t_k\}$ et soit $I^{'}$ l'interprétation $I\ [\![X_1/I(t_1), \ldots, X_k/I(t_k)]\!]$. Selon la règle de substitution, on applique :

$$I^{'} \not\models L_{m+1} \vee ... \vee L_p \vee L_{n+1}^{'} \vee ... \vee L_q^{'} \tag{3.2}$$

Mais puisque $S \models K_1$ et $S \models K_2$, il s'ensuit également que $S \models v_1(K_1)$ et $S \models v_2(K_2)$ et donc :

$$I^{'} \models L_1 \vee ... \vee L_m \vee L_{m+1} \vee ... \vee L_p \text{ et } I^{'} \models L_1^{'} \vee ... \vee L_n^{'} \vee L_{n+1}^{'} \vee ... \vee L_q^{'}$$

Avec (3.2), il s'ensuit que

$$I^{'} \models L_1 \vee ... \vee L_m \text{ et } I^{'} \models L_1^{'} \vee ... \vee L_n^{'}$$

Et avec le lemme de substitution, on obtient :

$$I \models \sigma(L_1 \vee ... \vee L_m) \text{ et } I \models \sigma(L_1^{'} \vee ... \vee L_n^{'})$$

On obtient donc la contradiction suivante :

$$I \models L \text{ et } I \models \neg L$$

Pour éviter cette contradiction, nous devons comprendre et suivre la résolution d'entrée et la SLD qu'on va essayer d'expliquer dans la section suivante.

3.5 Résolution d'entrée et SLD

Afin de réduire davantage les possibilités de résolution, l'une des clauses parentes doit être le dernier résolveur de chaque étape de résolution. L'autre clause parente pourrait cependant être choisie librement (c'est-à-dire qu'il pourrait s'agir d'une clause de l'ensemble de clauses d'origine ou d'une résolvante formée précédemment). Nous interdisons maintenant la dernière possibilité : maintenant une résolution doit être prise à chaque étape entre la dernière résolvante formée et l'une des clauses «d'entrée» d'origine. Pour cette raison, cette restriction est appelée résolution d'entrée CHEN et TSAI, 2014.

3.5.1 Définition (Entrée-Résolution)

Soit $\mathcal{K}$ un ensemble de clauses. La clause vide $\square$ peut être dérivée de la clause K dans $\mathcal{K}$ par la résolution d'entrée ssi. il existe une suite de clauses $K_1, ..., K_m$ telle que K_1 = K $\in \mathcal{K}$ et $K_m = \square$ et de sorte que pour tout 2 $\leq$ i $\leq$ m : K_i est une résolvante de K_{i+1} et une clause $\mathcal{K}$ LUGER et STUBBLEFIELD, 2008.

3.5.2 Exemple

Nous considérons à nouveau l'ensemble des clauses insatisfiables. En résolvant deux clauses à partir de l'ensemble original de clauses, seules les clauses suivantes peuvent être dérivées :

$$\{ q \}, \{ \neg q \}, \{ p \}, \{ \neg p \}, \{ q, \neg q \}, \{ p, \neg p \}$$

Si l'une des quatre premières clauses résultantes est à nouveau résolue avec une clause de l'ensemble d'origine, l'une des quatre premières clauses est de nouveau créée. Si vous résolvez l'une des deux dernières clauses (généralement valides) avec une clause de l'ensemble d'origine, vous obtenez la clause de l'ensemble d'origine. Une dérivation de la clause vide n'est possible que si deux des clauses résultant de la résolution (telles que { q } et { $\neg$ q }) sont résolues l'une avec l'autre. Cependant, cela n'est pas autorisé avec la résolution d'entrée. Bien que la résolution d'entrée ne soit complète sur aucun ensemble de clauses, elle est toujours complète sur l'ensemble restreint de clauses dites de Horn. Pour cette raison, aucune clause arbitraire n'est utilisée dans la programmation logique, seulement des clauses Horn.

3.5.3 Définition (Clause de Horn)

Une clause K est une clause Horn ssi. il contient au plus un littéral positif (c'est-à-dire qu'au plus l'un de ses littéraux est une formule atomique et les autres littéraux sont des formules atomiques négatives). Une clause de Horn est dite négative si elle ne contient que des littéraux négatifs (c'est-à-dire si elle a la forme $\{\neg A_1,..., \neg A_k\}$ pour les formules atomiques $A_1,..., A_k$). Une clause de Horn est dite définitive si elle contient un littéral positif (c'est-à-dire si elle a la forme $\{ B, \neg C_1,..., \neg C_n\}$ pour les formules atomiques $B, C_1, ..., C_n$) LLOYD, 1984.
Un ensemble de clauses de Horn définies correspond donc à une conjonction d'implications. Par exemple, la clause Horn

$$\{\{ p, \neg q \}, \{\neg r, \neg p, s \}, \{ s \}\}$$

équivalent à la formule

$$((p, \vee \neg q) \wedge (\neg r \vee \neg p \vee s) \wedge s)$$

et donc aussi à la formule suivante :

$$((q \rightarrow p) \wedge (r \wedge p \rightarrow s) \wedge s)$$

Vous pouvez voir la connexion à la programmation logique :

— Les faits sont des clauses de Horn définies sans littéraux négatifs (c'est-à-dire qu'ils contiennent exactement un littéral positif). Un exemple est la clause { s }. En programmation logique, on écrit :

s.

— Les règles sont des clauses de Horn définies avec des littéraux négatifs. Un exemple est la clause $\{\neg r, \neg p, s \}$. En programmation logique, on écrit :

s :- r, p.

— Les requêtes sont des clauses Horn négatives. Un exemple est la clause $\{\neg p, \neg q\}$. En programmation logique, on écrit :

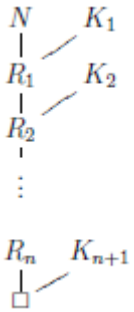
?- p, q.

La prise en compte des clauses Horn est une vraie limitation, car il n'y a pas un ensemble équivalent de clauses Horn pour chaque ensemble de clauses. Cela est déjà vrai en logique propositionnelle, car l'ensemble des clauses { p, q } en est un exemple. Dans la pratique, cependant, la restriction aux clauses Horn est souvent suffisante. L'avantage de cette restriction est que le test de satisfiabilité peut être effectué de manière beaucoup plus efficace et automatique. En logique propositionnelle, la satisfiabilité des ensembles de clauses de Horn est décidable en temps polynomial, tandis que le problème de satisfiabilité pour toute formule propositionnelle est connu pour être NP-complet ZOMBORI, URBAN et BROWN, 2020.
Dans le cas des clauses de Horn à logique de prédicat, l'(ir) remplissabilité est toujours indécidable. Mais la restriction aux clauses Horn entraîne également un gain d'efficacité significatif dans la logique des prédicats. La raison en est que nous pouvons maintenant nous limiter à la résolution d'entrée, car elle est complète aux ensembles de clauses Horn.
Avec les clauses Horn, la résolution d'entrée peut être étendue encore plus à la résolution SLD sans perdre l'exhaustivité.

3.5.4 Définition (Résolution-SLD)

Soit $\mathcal{K}$ une clause de Horn avec $\mathcal{K} = \mathcal{K}^d \uplus \mathcal{K}^n$, où $\mathcal{K}^d$ contient les clauses définies et $\mathcal{K}^n$ les clauses négatives de $\mathcal{K}$. La clause vide $\square$ peut être dérivée de la clause K dans $\mathcal{K}^n$ par la résolution SLD ssi. il existe une suite de clauses $K_1, ..., K_m$ telle que K_1 = K $\in \mathcal{K}^n$ et $K_m = \square$ et de sorte que pour tout 2 $\leq$ i $\leq$ m : K_i est une résolvante de K_{i+1} et une clause $\mathcal{K}^d$ LLOYD, 1984.

Vous pouvez voir immédiatement que toutes les clauses $K_1, ..., K_m$ dans une résolution SLD sont négatives. Les résolutions SLD ont donc la forme suivante :



Ici $K_1, ..., K_{n+1} \in \mathcal{K}^d$ sont des clauses de Horn définies de l'ensemble d'entrée, N $\in \mathcal{K}^n$ est une clause de Horn négative de l'ensemble d'entrée et les résolvantes $R_1, ..., R_n$ sont également des clauses de Horn négatives.

L'abréviation « SLD » signifie « résolution linéaire avec fonction de sélection pour les clauses définies ». Dans l'étape de résolution de N à R_1 ou dans l'étape de résolution de R_{i-1} à R_i, la « fonction de sélection » doit sélectionner les littéraux de N ou R_{i-1} avec lesquels résoudre. Nous ignorons actuellement cette fonction de sélection et autorisons la résolution avec tous les littéraux de la définition ci-dessus. (En ce moment, nous examinons la « résolution LUSH », où « LUSH » signifie « résolution linéaire avec sélection illimitée pour les clauses Horn »). En plus du choix du littéral à résoudre, l'autre option pour la résolution SLD est le choix de la clause d'entrée à utiliser à partir de $\mathcal{K}^d$. La manière dont ces deux options sont davantage restreintes dans la programmation logique est discutée plus en détail plus loin dans ce cours. La phrase suivante montre l'exhaustivité de la résolution SLD sur les clauses Horn.

3.5.5 Théorème (exactitude et exhaustivité de la résolution SLD)

Soit $\mathcal{K}$ un ensemble de clauses de Horn. Alors $\mathcal{K}$ est insatisfiable ssi. $\square$ peut être dérivée d'une clause négative N dans $\mathcal{K}$ par résolution SLD APT, 2001.

Preuve.

L'exactitude (c'est-à-dire la direction « $\Leftarrow$ ») est à nouveau évidente, puisque chaque étape de résolution SLD est également une étape de résolution. Nous montrons donc maintenant la complétude (c'est-à-dire la direction « $\Rightarrow$ ». Soit $\mathcal{K}_{min}$ un sous-ensemble minimal insatisfiable de $\mathcal{K}$. Tout ensemble de clauses de Horn définies est satisfiable, car l'interprétation qui satisfait toutes les formules atomiques est également un modèle de chaque clause de Horn définie. Par conséquent, $\mathcal{K}_{min}$ doit également contenir une clause de Horn négative N. La raison en est que toutes les clauses négatives résolvantes et que les clauses négatives ne peuvent pas être résolues avec des clauses négatives (uniquement des clauses définies).

L'algorithme semi-décidal pour vérifier l'inférence ou l'insatisfiabilité peut maintenant être amélioré. Si nous recevons un ensemble de clauses composé uniquement de clauses Horn, nous ne calculons que toutes les séquences de résolution SLD, mais nous devons commencer par toute clause négative.
La résolution SLD pour les ensembles de clauses Horn constitue la base opérationnelle de la programmation logique, comme cela apparaîtra clairement dans le chapitre suivant. Cependant, on se limite au fait qu'à chaque étape de résolution on ne résout qu'entre deux littéraux, pas entre un nombre quelconque. Dans la définition précédente de la résolution logique du prédicat, les littéraux $L_1, ..., L_m$ de la première clause parent et les littéraux $L_1^{'}, ..., L_n^{'}$ de la deuxième clause parent sont supprimés si $\{\neg L_1, ..., \neg L_m, L_1^{'}, ..., L_n^{'}\}$ est unifiable. Au lieu de cela, nous utilisons maintenant une contrainte où m = n = 1. C'est ce qu'on appelle la résolution binaire BRATKO, 2011.

3.5.6 Exemple

Un contre-exemple de l'exhaustivité de la résolution binaire est l'ensemble de clauses suivant. Ici p $\in \Delta_1$.

$$\{\{\text{ p(X), p(Y)}\}, \{\neg\text{ p(U), }\neg\text{ p(V)}\}\}$$

En utilisant les pgu { X/V, Y/V, U/V } de l'ensemble { p(X), p(Y), p(U), p(V) }, la clause vide peut être dérivée en une étape avec une résolution logique de prédicat. L'ensemble des clauses est donc insatisfiable. Avec la résolution binaire, on ne peut dériver que des clauses comme { p(Y), ¬ p(V) }. Si vous résolvez deux de ces clauses, vous en obtenez une autre. Si on résout une de ces clauses avec une clause d'entrée, alors on obtient une clause qui est équivalente à la clause d'entrée. On ne peut donc pas dériver la clause vide. Cependant, la clause { p(X), p(Y)} n'est pas non plus une clause de Horn.

3.5.7 Théorème (exactitude et exhaustivité de la résolution SLD binaire)

Soit $\mathcal{K}$ un ensemble de clauses de Horn. Alors $\mathcal{K}$ est insatisfiable ssi. □ peut être dérivée d'une clause négative N dans $\mathcal{K}$ en utilisant la résolution SLD binaire LLOYD, 1984.

Preuve.

L'exactitude (c'est-à-dire la direction « ⇐ ») est à nouveau évidente, puisque chaque étape de résolution SLD binaire est également une étape de résolution. Nous montrons donc maintenant la complétude (c'est-à-dire la direction « ⇒ ». Nous faisons cela en deux étapes. Tout d'abord, nous montrons que chaque étape de résolution avec une résolution SLD générale peut être remplacée par une séquence d'étapes de résolution SLD binaires illimitées. La résolution SLD illimitée signifie que vous pouvez utiliser n'importe quel unificateur au lieu de pgu. Nous prouvons ensuite que pour chaque preuve d'insatisfiabilité avec une résolution SLD binaire illimitée, il existe également une preuve d'insatisfiabilité avec une résolution SLD binaire. Puisque la résolution SLD générale sur les clauses de Horn est complète, l'assertion de la proposition suit.
Nous montrons tout d'abord que chaque étape de résolution avec une résolution SLD générale peut être remplacée par une séquence d'étapes de résolution SLD binaires illimitées. Une étape de résolution SLD générale résout une clause de Horn négative N = $\{\neg A_1, ..., \neg A_m, \neg A_{m+1}, ..., \neg A_p\}$ avec une clause définie comme suit K

= $\{B, \neg C_1, ..., \neg C_n\}$ en utilisant le pgu σ de l'ensemble $\{A_1, ..., A_m, v_1$ (B) $\}$ à la résolvante R = $\sigma(\{\neg A_{m+1}, ..., \neg A_p, \neg\ v_1\ (C_1)\ ,..., \neg\ v_1\ (C_n)\})$. Ici, v_1 est un renommage de variable approprié. Puisque σ est pgu de $\{A_1, ..., A_m, B\}$, σ est en particulier aussi un unificateur de A_1 et B (mais pas nécessairement leur pgu). Avec une résolution SLD binaire illimitée, et $\mathcal{K}$ dérivent ainsi la résolvante.

$$R_1 = \sigma(\{\neg A_2, ..., \neg A_m, \neg A_{m+1}, ..., \neg A_p, \neg v_1\ (C_1),..., \neg v_1\ (C_n)\})$$

Maintenant, nous utilisons une autre variable renommant v_2 et développons σ de telle sorte que $\sigma\circ\ v_2 = \sigma\circ\ v_1$. Cela est possible parce que v_2 n'introduit que de nouvelles variables qui n'étaient pas du domaine de σ. Alors σ est l'unificateur de $\sigma(A_2)$ et $v_2\ (B)$ et par une résolution SLD binaire non bornée renouvelée de R_1 et $\mathcal{K}$ on peut obtenir la résolvante

$$R_2 = \sigma(\{\neg A_3, ..., \neg A_m, \neg A_{m+1}, ..., \neg A_p, \neg v_1\ (C_1),..., \neg v_1\ (C_n)\})$$

puisque $\sigma(\sigma(A_i)) = \sigma(A_i)$ et $\sigma(\sigma\ (v_1\ (C_i))) = \sigma\ (v_2\ (C_i)))$. Globalement, après m étapes de résolution binaire non bornées, on obtient la clause

$$R = \sigma(\{\neg A_{m+1}, ..., \neg A_p, \neg v_1\ (C_1),..., \neg v_1\ (C_n)\}$$

Nous prouvons maintenant que chaque preuve de résolution SLD binaire illimitée de la clause vide avec n étapes peut également être transformée en une preuve de résolution SLD binaire. Pour ce faire, nous utilisons l'induction sur n. Dans le cas n = 0, c'est trivial. Sinon, une clause de Horn négative N = $\{\neg A_1, \ldots, \neg A_m\}$ avec une clause définie $K_1 = \{B, \neg C_1, ..., \neg C_p\}$ en utilisant l'unificateur (pas nécessairement le plus général) σ_1 de l'ensemble $\{A_1, v_1\ (B)\}$ pour la résolvante

$$R_1 = \sigma_1(\{\neg A_2, ..., \neg A_m, \neg v_1\ (C_1),..., \neg v_1\ (C_p)\}).$$

Si $R_1 = \square$, on aurait évidemment pu utiliser le pgu correspondant à la place de l'unificateur σ_1. Sinon, l'étape suivante est résolue avec une clause définie $K_2 = \{D, \neg E_1, ..., \neg E_q\}$. Ici σ_2 est un unificateur de $\sigma_1(A_2)^2$ et v_2 (D). Alors la résolvante

$$R_2 = \sigma_2(\sigma_1(\{\neg A_3, ..., \neg A_m, \neg v_1\ (C_1),..., \neg v_1\ (C_p)\}) \cup \{\neg v_2\ (E_1),...,\neg v_2\ (E_q)\})$$

Soit θ upg de $\{A_1, v_1\ (B)\}$, alors il y a une substitution σ avec $\sigma_1 = \delta \circ \theta$. Avec la résolution SLD binaire, la clause

$$R_1^{'} = \theta(\{\neg A_2, ..., \neg A_m, \neg v_1\ (C_1),..., \neg v_1\ (C_p)\})$$

qui a été obtenue à partir de N et K_1. Avec une autre étape de résolution SLD binaire illimitée, nous avons pu résoudre $R_1^{'}$ et K_2 l'un avec l'autre. Pour unifier $\theta(A_2)$ et v_2 (D), on peut utiliser l'unificateur $\sigma_2 \circ \delta$, car $\sigma_2(\delta(\theta(A_2))) = \sigma_2(\sigma_1(A_2)) = \sigma_2(v_2$ (D)) = $\sigma_2(\delta(v_2\ (D)))$ (puisque le domaine de δ ne contient aucune variable introduite par v_2) Le résolvant R_2 résulte à nouveau de la résolution SLD binaire illimitée de $R_1^{'}$ et K_2. On peut donc dériver la clause vide en n-1 étapes de $R_1^{'}$ à la résolution SLD binaire illimitée. Selon l'hypothèse d'induction, cela est alors également possible grâce à la résolution SLD binaire. Puisque la première étape de N à $R_1^{'}$ est également une étape de résolution SLD binaire, vous pouvez faire la preuve entière avec une résolution SLD binaire.

Chapitre 4

Programmes logiques

4.1 Introduction

Dans ce chapitre, nous présentons maintenant formellement les programmes de logique (pure) et définissons leur syntaxe et leur sémantique dans la section suivante. Ceci est bien sûr basé sur les principes fondamentaux de la logique des prédicats et sur le calculateur de résolution des chapitres précédents. Puis, nous montrons que la programmation logique est universelle, c'est-à-dire que l'on peut en fait calculer tous les programmes calculables avec ce langage de programmation.

4.2 Syntaxe et sémantique des programmes logiques

La définition suivante introduit formellement les programmes logiques. Ici, nous utilisons à nouveau un ensembles de clauses Horn. Contrairement au chapitre précédent, l'ordre des littéraux dans une clause et l'ordre des clauses dans un ensemble de clauses jouent désormais un rôle. Nous considérons donc désormais des séquences au lieu d'ensembles. Ainsi, lorsque nous parlons de « clauses », nous entendons des séquences de littéraux et lorsque nous parlons d'« ensembles de clauses », nous entendons des séquences de clauses. Nous conservons la notation précédente, c'est-à-dire que nous continuons à écrire des clauses et des ensembles de clauses avec des crochets. Cependant, l'ordre des littéraux ou des clauses n'est plus sans importance et une clause peut également contenir un littéral plus d'une fois et un ensemble de clauses peut contenir une clause plus d'une fois BRATKO, 2011.

4.2.1 Définition (Syntaxe des programmes logiques)

Un ensemble fini non vide $\mathcal{P}$ de clauses de Horn définies sur une signature (Σ, Δ) est appelé un programme logique sur (Σ, Δ). Les clauses de $\mathcal{P}$ sont également appelées clauses de programme et une distinction est faite entre les types de clauses de programme suivants CHEN et TSAI, 2014 :

— Les faits (ou « clauses factuelles ») sont des clauses de la forme $\{B\}$, où B est une formule atomique

— Les règles (ou dites « clauses de procédure ») sont un ensemble de clauses de la forme $\{B, \neg C_1, ..., \neg C_n\}$ avec n ≥ 1 où B et $C_1, ..., C_n$ sont des formules atomiques.

— But (ou « clause cible ») G de la forme $\{\neg A_1, ..., \neg A_k\}$ avec k ≥ 1
Comme précédemment, une clause représente la disjonction tout quantifiée de ses littéraux et un ensemble de clauses (comme un programme logique)

représente la conjonction de ses clauses. Lors de l'appel d'un programme logique $\mathcal{P}$ avec la requête G = $\{\neg A_1, ..., \neg A_k\}$, ce qui suit doit être prouvé :

$$\mathcal{P} \models \exists X_1, ..., X_p \; A_1 \wedge ... \wedge A_k \tag{4.1}$$

Ici $X_1, ..., X_p$ sont les variables de G. Les variables des clauses du programme sont implicitement toutes quantifiées et les variables des requêtes sont implicitement quantifiées existentiellement. La relation d'inférence ci-dessus équivaut à l'insatisfiabilité de l'ensemble des clauses $\mathcal{P} \cup \{G\}$, c'est-à-dire à l'insatisfiabilité de $\mathcal{P} \cup \{\forall X_1, ..., X_p \neg A_1 \vee ... \vee \neg A_k\}$.
L'insatisfiabilité de $\mathcal{P} \cup \{G\}$ équivaut au fait qu'il existe un sous-ensemble fini d'instances de base des clauses de $\mathcal{P} \cup \{G\}$ qui est également insatisfiable. Cet ensemble ne peut pas seulement être constitué de clauses définies, c'est-à-dire qu'il contient également au moins une instance de base de G. Par contre, à partir de l'exhaustivité de la résolution SLD pour les ensembles de clauses de Horn, nous savons que la clause vide peut être dérivée d'un ensemble insatisfaisable de clauses Horn, une seule clause négative est nécessaire. Ainsi, l'insatisfiabilité de $\mathcal{P} \cup \{G\}$ signifie qu'il existe un ensemble fini non satisfaisable d'instances de base des clauses de $\mathcal{P} \cup \{G\}$ qui contient exactement une instance de base de G. Donc (4.1) équivaut au fait qu'un ensemble de termes fondamentaux $t_1, ..., t_p$, existe de telle sorte que

$\mathcal{P} \cup \{(\neg A_1 \vee ... \vee \neg A_k)[X_1/t_1, ..., X_p/t_p]\}$ est insatisfaisable ou que
$\mathcal{P} \models A_1 \wedge ... \wedge A_k[X_1/t_1, ..., X_p/t_p]$

le but n'est pas seulement d'étudier la relation d'inférence de (4.1), mais aussi d'examiner les termes de base $t_1, ..., t_p$, qui sont les « solutions » valides pour la requête. Les substitutions dans lesquelles la requête souhaitée découle des clauses du programme sont appelées substitution de réponse. La substitution est limitée aux variables de la requête. Si la substitution de réponse remplace les variables de la requête par des termes avec des variables, les variables restantes peuvent être remplacées par n'importe quel terme. Les substitutions de réponses sont générées au cours de la preuve de résolution SLD lorsque la clause vide $\square$ est dérivée BRAMER, 2013.

4.2.2 Exemple

Nous considérons un sous-ensemble du programme logique du chapitre 1 :

```
mere(souad, hoda).
marie(ali, souad).
pere(V, K)    :- marie(V, F), mere(F, K).
```

Si nous écrivons ce programme logique $\mathcal{P}$ comme un ensemble de clauses, nous obtenons

$$\begin{array}{ll} \{ & \{ \text{ mere(souad, hoda)}\}, \\ & \{ \text{ marie(ali, souad)}\}, \\ & \{ \text{ pere(V, F), } \neg \text{ marie(V, F), } \neg \text{ mere(F, K)}\} \; \}. \end{array}$$

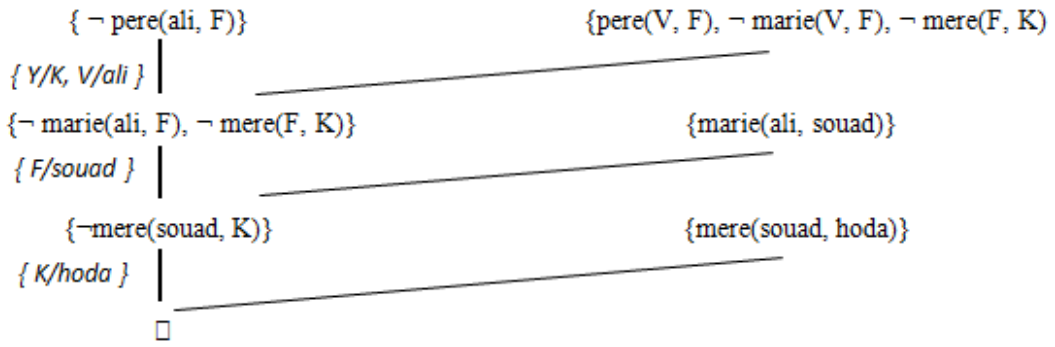
Nous étudions la requête

```
?- pere(ali, Y).
```

Cela signifie qu'en plus de l'ensemble des clauses de Horn définies ci-dessus, nous ajoutons la clause de Horn négative G. Ajoutez

{¬ pere(ali, Y) }.

Nous obtenons maintenant la preuve de résolution SLD suivante pour déduire la clause vide. Le upg qui a été appliqué à la clause parent négative et à la clause programme (éventuellement renommée en tant que variable) a été spécifié à chaque étape de résolution.



La substitution est maintenant obtenue en composant les substitutions individuelles

{ **K/hoda** } ∘ {**F/souad** } ∘ { **Y/K, V/ali** } = { **K/hoda, F/souad, Y/hoda, V/ali** }

puis le restreint aux variables de la demande. Puisque la requête ne contient que la variable Y dans notre exemple, la substitution de réponse est $\{Y/hoda\}$. Nous définissons maintenant la sémantique des programmes logiques. Ici, nous allons introduire trois manières différentes de spécifier la sémantique et nous prouverons que les trois manières sont équivalentes. Ces types sont connus sous le nom de sémantique déclarative, procédurale et à virgule fixe COLMERAUER, 1990.

4.2.3 Sémantique déclarative de la programmation logique

L'idée de la sémantique déclarative (ou « théorique du modèle ») est de comprendre le programme logique comme une « base de données » statique et de définir les vraies déclarations sur le programme à l'aide de l'inférence de la logique des prédicats. Plus précisément, nous définissons la sémantique d'un programme $\mathcal{P}$ par rapport à une requête G. La sémantique déclarative est constituée de toutes les instances de base de G qui sont des « déclarations vraies » sur le programme LLOYD, 1984.

4.2.4 Définition (sémantique déclarative d'un programme logique)

Soit $\mathcal{P}$ un programme logique et G = $\{\neg A_1, ..., \neg A_k\}$ une requête. Ici $A_1, ..., A_k$ sont des formules atomiques. Alors la sémantique déclarative de $\mathcal{P}$ par rapport à G est définie comme NILSSON et MALUSZYNSKI, 1995 :

$$D[\![\mathcal{P}, G]\!] = \{\sigma(A_1 \wedge ... \wedge A_k) | \mathcal{P} \models \sigma(A_1 \wedge ... \wedge A_k)\}$$

Avec σ est la substitution de base. Chaque instance de base$\sigma(A_1 \wedge ... \wedge A_k)$ dans $D[\![\mathcal{P}, G]\!]$ contient comme « solution » la substitution de base correspondante des variables de $A_1, ..., A_k$.

4.2.5 Exemple

Nous considérons le programme logique $\mathcal{P}$ et la requête G = $\{\neg pere(ali, Y)\}$ de l'exemple passé. La seule instance de base de pere(ali, Y) qui découle de $\mathcal{P}$ est pere(ali, hoda) (c'est-à-dire $\mathcal{P} \models$ pere(ali, hoda)). D'où

$$D[\![\mathcal{P}, G]\!] = \{pere(ali, hoda)\}$$

Si $\mathcal{P}$ contenait le fait supplémentaire mere(souad, omar), le résultat serait

$$D[\![\mathcal{P}, G]\!] = \{pere(ali, hoda), pere(souad, omar)\}$$

4.2.6 Sémantique procédurale de la programmation logique

La sémantique procédurale (ou « opérationnelle ») « opérationnalise » la sémantique déclarative en spécifiant explicitement comment les inférences correspondantes de $\mathcal{P}$ sont calculées. À cette fin, la résolution SLD est utilisée, qui enregistre également les substitutions appliquées aux variables. Plus précisément, nous donnons un interpréteur abstrait pour les programmes logiques qui opèrent sur des configurations. Une configuration est une paire d'une demande (c'est-à-dire une clause négative) et une substitution. Ici, nous commençons par la configuration $(G, \emptyset)$ de la requête d'origine et la substitution identique $\emptyset$. Le but est d'arriver enfin à une configuration finale de la forme $(\square, \sigma)$. Dans ce cas, σ (ou la restriction de σ aux variables de la requête originale G) est la substitution de réponse trouvée. Un calcul est une séquence de configurations, avec la définition de l'interpréteur définissant les transitions de configuration autorisées NILSSON et MALUSZYNSKI, 1995.

La forme de la résolution SLD utilisée dans les programmes logiques diffère en trois points de la définition de la résolution SLD générale.

— Au lieu d'appliquer un changement de nom de variable aux deux clauses parentes, nous nous limitons à la résolution SLD dite normalisée. Le changement de nom de variable ne peut être appliqué qu'aux clauses de programme (définies) et non à l'autre clause parent (négative). Une telle restriction est bien entendu généralement possible sans restreindre le grand public.

— Dans l'étape de résolution, seuls deux littéraux sont résolus, pas entre un nombre quelconque. Nous n'utilisons donc que la résolution SLD binaire. Cependant, cette restriction sur les ensembles de clauses Horn est complète.

— Après tout, nous ne considérons plus les clauses comme des ensembles, mais comme des séquences de littéraux. Cela signifie qu'un littéral peut apparaître plusieurs fois dans une telle séquence. Par exemple, la résolution des clauses $\{\neg p, \neg p\}$ et $\{p\}$ aboutit à la résolvante $\{\neg p\}$. Ici, $p \in \Delta_0$ s'applique comme à la preuve de l'exhaustivité de la résolution SLD binaire, l'exhaustivité de la résolution SLD binaire suit immédiatement ces séquences. La raison est en que chaque étape de résolution précédente sur les ensembles peut évidemment être remplacée par une séquence d'étapes de résolution sur les séquences.

4.2.7 Définition (sémantique procédurale d'un programme logique)

Soit $\mathcal{P}$ un programme logique.

— Une configuration est un couple (G, σ), où G est une requête ou la clause vide $\square$ et où σ est une substitution.

— Il y a une étape de calcul $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ ssi.

- $G_1 = \{\neg A_1..., \neg A_k\}$ avec $k \geq 1$
- Il existe une clause de programme $K \in \mathcal{P}$ et une variable v avec $v(K) = \{B, \neg C_1, ..., \neg C_n\}$ et $n \geq 0$, de sorte que
  * $v(K)$ n'a pas de variables communes avec G_1 ou $RANGE(\sigma_1)$ et
  * Il existe un $1 \leq i \leq k$ tel que A_i et B sont unifiables avec un pgu σ
- $G_2 = \sigma(\{\neg A_1, ..., \neg A_{i-1}, \neg C_1, ..., \neg C_n, \neg A_{i+1}, ..., \neg A_k\})$
- $\sigma_2 = \sigma \circ \sigma_1$

— Un calcul de $\mathcal{P}$ en entrant $G = \{\neg A_1, ..., \neg A_k\}$ est une suite (finie ou infinie) de configurations de la forme

$$(G, \emptyset) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} ...$$

— Un calcul se terminant par $(\square, \sigma)$ et commençant par $(G, \emptyset)$ (où $G = \{\neg A_1, ..., \neg A_k\}$) est dit réussi avec le résultat du calcul $\sigma(A_1 \wedge ... \wedge A_k)$. La substitution de réponse calculée est restreinte σ aux variables de G. Ainsi, la sémantique procédurale de $\mathcal{P}$ par rapport à G est définie comme

$P[\![\mathcal{P}, G]\!] = \{\sigma'(A_1 \wedge ... \wedge A_k) | (G, \emptyset) \vdash^+_{\mathcal{P}} (\square, \sigma), \sigma'(A_1 \wedge ... \wedge A_k)$ est l'instance de base de $\sigma(A_1 \wedge ... \wedge A_k)\}$

Ici $\vdash^+_{\mathcal{P}}$ représente l'enveloppe transitive de « $\vdash_{\mathcal{P}}$ » (c'est-à-dire $(G, \emptyset) \vdash^+_{\mathcal{P}} (\square, \sigma)$ ssi $(G, \emptyset) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma)$. De manière analogue, on définit aussi pour tout $l \in \mathbb{N}$ la relation $\vdash^l_{\mathcal{P}}$ comme $(G, \sigma) \vdash^l_{\mathcal{P}} (G_l, \sigma_l)$ ssi. il y a G_i et σ_i avec $(G, \sigma) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} ... \vdash_{\mathcal{P}} (G_l, \sigma_l)$

4.2.8 Exemple

La sémantique procédurale correspond exactement à la procédure de l'exemple 4.1.2. Ici s'applique :

$$\begin{array}{ll} & (\{\neg \text{ pere(ali, Y)}\}, \emptyset) \\ \vdash_{\mathcal{P}} & (\{\neg \text{ marie(ali, F)}, \neg \text{ mere(F, K)}\}, \{\text{Y/K, V/ali}\}) \\ \vdash_{\mathcal{P}} & (\{\neg \text{ mere(souad, K) }\}, \{\text{F/souad, Y/K, V/ali}\}) \\ \vdash_{\mathcal{P}} & (\square, \{\text{K/hoda, F/souad, Y/hoda, V/ali}\}) \end{array}$$

et la substitution de réponse est $\{Y/hoda\}$. Nous obtenons

$$P[\![\mathcal{P}, G]\!] = \{ \text{ pere(ali,hoda) } \}$$

L'exemple suivant montre qu'il existe deux autres indéterminismes dans les étapes de calcul de la sémantique procédurale :

— D'une part, vous devez sélectionner la clause de programme K avec laquelle la résolution doit être effectuée.

— D'un autre côté, vous devez sélectionner le littéral A_i suivant de la requête courante à utiliser pour la résolution.

4.2.9 Exemple

Nous considérons le programme logique suivant $\mathcal{P}$:

$$\begin{array}{ll} \{ & \{ \text{ p(X, Z), } \neg \text{ q(X, Y), } \neg \text{ p(Y, Z)}\}, \\ & \{ \text{ p(U, U)}\}, \\ & \{ \text{ q(a, b) } \} \ \}. \end{array}$$

Avec p, q $\in \Delta_2$ et a, b $\in \Sigma_0$. La requête à examiner est

$$G = \{\neg p(V, b)\}.$$

Le calcul suivant est infructueux (mais fini et ne peut plus être poursuivi). Le littéral A_i utilisé pour la résolution est souligné :

$$\begin{array}{ll} & (\underline{\{\neg p(V,b)\}}, \emptyset) \\ \vdash_{\mathcal{P}} & (\underline{\{\neg q(V,Y), \neg p(Y,b)\}}, \{X/V, Z/b\}) \\ \vdash_{\mathcal{P}} & (\underline{\{\neg p(b,b)\}}, \{V/a, Y/b\} \circ \{X/V, Z/b\}) \\ \vdash_{\mathcal{P}} & (\{\neg q(b,Y^{'}), \underline{\neg p(Y^{'},b)}\}, \{X^{'}/b, Z^{'}/b\} \circ \{V/a, Y/b\} \circ \{X/V, Z/b\}) \\ \vdash_{\mathcal{P}} & (\{\neg q(b,b)\}, \underline{\{U/b, Y^{'}/b\}} \circ \{X^{'}/b, Z^{'}/b\} \circ \{V/a, Y/b\} \circ \{X/V, Z, b\}) \end{array}$$

Un calcul réussi, cependant, serait le suivant, qui a les mêmes trois premières étapes, mais se résout ensuite avec la seconde au lieu de la première clause de programme :

$$\begin{array}{ll} & (\underline{\{\neg p(V,b)\}}, \emptyset) \\ \vdash_{\mathcal{P}} & (\underline{\{\neg q(V,Y)}, \neg p(Y,b)\}, \{X/V, Z/b\}) \\ \vdash_{\mathcal{P}} & (\underline{\{\neg p(b,b)\}}, \{V/a, Y/b\} \circ \{X/V, Z/b\}) \\ \vdash_{\mathcal{P}} & (\square, \underbrace{\underline{\{U/b\}} \circ \{V/a, Y/b\} \circ \{X/V, Z/b\}}_{\{\mathrm{U/b\ ,\ V/a\ ,\ Y/b\ ,\ X/a\ ,\ Z/b}\}}) \end{array}$$

La substitution est { V/a } et donc p(a, b) $\in P[\![\mathcal{P}, G]\!]$. Dans cet exemple, il y a un autre calcul réussi, dans lequel on résout avec la deuxième clause de programme dans la première étape :

$$\begin{array}{ll} & (\underline{\{\neg p(V,b)\}}, \emptyset) \\ \vdash_{\mathcal{P}} & (\square, \{U/b, V/b\}) \end{array}$$

La substitution est maintenant { V/b } et donc p(b, b) $\in P[\![\mathcal{P}, G]\!]$ est également valable.

dans le chapitre prochain, nous allons voir au plus prés le langage prolog.

Chapitre 5

Le langage de programmation Prolog

5.1 Introduction

Maintenant que nous avons appris les bases de la programmation logique, considérons un langage de programmation spécifique basé sur ce principe. Le langage de programmation logique le plus populaire est Prolog. Ce langage a été développé par Kowalski et Colmerauer dans la première moitié des années 1970 COLMERAUER, 1990. Son acceptation comme l'un des langages essentielles de l'IA est venue du fait que Prolog a été choisi comme un langage principal pour le projet japonais de cinquième génération en 1981 ROWE, 1988.

La syntaxe des programmes Prolog (simples) correspond exactement à la syntaxe des programmes logiques. La notation bien connue avec " :-" pour les règles et avec "?-" pour les requêtes est utilisée. La signature d'un programme Prolog résulte des symboles de fonction et de prédicat qui apparaissent, chacun devant commence par une lettre minuscule. De plus, les chaînes composées de caractères spéciaux (par exemple $<--->$) et de chaînes entre guillemets (par exemple "X") sont également autorisées. Les variables commencent par une lettre majuscule ou un trait de soulignement. Une particularité est la variable anonyme "_ ". Les occurrences multiples de cette variable sont considérées comme différentes et dans le cas des substitutions, les affectations de cette variable ne sont pas incluses. Pour un programme avec le fait "p(a, b, c)." la requête est "?- p(_, _, X)." qui donne la substitution X = c.

Prolog permet la surcharge des symboles de fonction et de prédicat. Le programme ci-dessus avec le fait « p(a, b, c). » pourrait donc par exemple être complété par le fait « p(a, b). » Ici, **p** est un autre symbole de prédicat à deux paramètres qui n'a rien à voir avec le symbole de prédicat p à trois paramètres. Le programme ci-dessus pourrait également être complété par le fait « p(p(a, b), c, c) ». Maintenant, le **p** intérieur est un symbole de fonction à deux paramètres qui est indépendant du symbole de prédicat extérieur **p** à trois paramètres. Puisque les symboles avec une arité différente sont considérés comme différents, leur arité est souvent donnée après leur nom avec une barre oblique (c'est-à-dire **p**/2 et **p**/3).

La sémantique de Prolog correspond à la sémantique des programmes logiques du chapitre précédent. L'arborescence SLD est parcourue dans le cadre d'une recherche en profondeur. Prolog s'arrête à la première substitution trouvée. Si l'utilisateur entre ensuite « ; », l'arborescence SLD est recherchée plus loin jusqu'à l'occurrence suivante de la clause vide, etc. . La raison est que le programmeur doit être clair sur la stratégie d'évaluation de Prolog et doit donc, par exemple, écrire des faits avant les clauses récursives afin que Prolog trouve réellement des solutions. La logique

n'est pas complètement séparée du contrôle de l'exécution du programme.

Cependant, une différence importante par rapport à la sémantique des programmes logiques du chapitre précédent est que la plupart des implémentations Prolog n'effectuent pas de contrôle d'occurrence pendant l'unification pour des raisons d'efficacité. Donc, si une variable X doit être unifiée avec un terme partiel $t \neq X$, aucun contrôle n'est effectué pour savoir si X apparaît dans le terme t. Au lieu de cela, une référence à la cellule mémoire correspondant au terme t est écrite dans la cellule mémoire correspondant à la variable X. Si l'on veut unifier X avec le terme f(X), il en résulte une substitution dans laquelle chaque occurrence de X se voit attribuer le terme instancié correspondant f(X). Donc, vous obtenez la substitution $\{X/f(f(f(...)))\}$ en tant qu'unificateur. On attribue donc à la variable X le terme infini constitué de rien d'autre que des symboles f. Une telle substitution est obtenue, par exemple, à partir du programme avec le fait « p(X, f(X)) ». et la requête « ?- p(X, X). ».

Prolog dispose de nombreux prédicats prédéfinis, dont certains sont également présentés ci-dessous. En particulier, il existe également un prédicat prédéfini appelé *unify_with_occur_check*, qui effectue une unification correcte (avec un contrôle d'occurrence). La requête « ?- unify_*with_occur*_check(X, f(X)). » la réponse est FALSE. En revanche, la requête aboutit à « ?- unify_*with_occur*_check(X, f(Y)). » la substitution X = f(Y).

Étant donné que les programmes logiques ne fonctionnent que sur des termes en tant que données, les objets de données doivent être représentés sous forme de termes à l'aide de symboles de prédicat appropriés. Pour certaines structures de données (en particulier les nombres entiers et les listes (linéaires)) il existe cependant certaines notations et support dans Prolog afin d'améliorer l'efficacité et la lisibilité des programmes. Le traitement de l'arithmétique et des listes dans Prolog serai présenté. Ensuite, nous verrons comment l'utilisateur peut définir d'autres opérateurs en plus des opérateurs arithmétiques intégrés (c'est-à-dire des symboles de fonction avec une notation infixe, préfixe ou postfixe). Nous avons introduit le prédicat de coupe prédéfini, qui est utilisé pour élaguer l'arbre SLD, et enfin, nous montrons comment la négation peut être implémentée de cette manière LLOYD, 1984.

5.2 L'arithmétique dans Prolog

Les nombres naturels peuvent être représentés sous forme de termes sur les symboles de prédicat $0 \in \Sigma_0$ et $s \in \Sigma_1$. Nous avons voir le programme suivant pour l'addition. (Ici, nous utilisons le symbole de prédicat **add**, car **plus** est déjà prédéfini dans Prolog.) **add(X, 0, X). add(X, s(Y), s(Z)) :- add(X, Y, Z).** Ici, « **add(X, Y, Z)** » signifie l'instruction « **X + Y = Z** ». Donc, si vous faites une requête dans laquelle les deux premiers arguments sont donnés par **add**, ce programme fait l'addition. L'appel « **?- Add(s(0), s(s(0)), X).** » donne donc la réponse **X = s(s(s(0)))**.

Cependant, comme il n'y a pas d'arguments d'entrée et de sortie fixes dans les programmes logiques, ce programme peut également être utilisé pour la soustraction, par exemple en définissant les deuxième et troisième arguments. Ce comportement est également appelé bidirectionnalité. Pour calculer « 3-2 », posez la requête « **?- Add(X, s(s(0)), s(s(s(0)))).** » et vous obtenez la réponse **X = s(0)**.

Vous pouvez également ajouter un nombre en demandant « **?- add(X, Y, s(s(s(0)))).** » calculer toutes les paires de sommations. La requête « **?-Add(X, s(s(0)), Z)** » fournit

également un résultat significatif (toutes les paires de valeurs, de sorte que la première valeur soit 2 plus petite que la seconde, c'est-à-dire **Z = s(s(X))**. En revanche, il existe un nombre infini de substitutions de réponses pour la requête « **?- Add(s(0), Y, Z)** ». Cela montre que pour presque tous les programmes Prolog, il existe également des requêtes qui conduisent à la non-terminaison ou à une arborescence SLD infinie.

Un inconvénient de la représentation des nombres naturels sous forme de termes supérieurs à 0 et s est que cela conduit souvent à des programmes inefficaces et difficiles à lire (en raison du manque d'opérations arithmétiques de base et de la taille des termes). Pour cette raison, Prolog autorise la notation habituelle pour les nombres entiers et fournit des fonctions de base et des prédicats prédéfinis.

Une expression arithmétique est un terme qui est construit de manière inductive à partir de nombres, de variables et de fonctions telles que $+, -, *, //$(division entière), $**$ (pour la puissance), etc. Comme précédemment, ces expressions peuvent être comprises comme des termes et traitées avec une unification syntaxique normale. Donc, si vous avez un programme avec le fait « **equal(X, X).** » a, la requête « **?- equal(3,1 + 2).** » renvoie faux. La requête « **?- equal(X, 1 + 2).** » donne la solution **X = 1 + 2**. Alors que seule l'unification syntaxique normale est utilisée dans l'évaluation normale des requêtes, il existe d'autres prédicats prédéfinis qui utilisent les implémentations prédéfinies correspondantes des fonctions $+, -, *, //, **$, etc.

Pour comparer les expressions arithmétiques, Prolog propose, entre autres, les prédicats à deux paramètres op avec $op \in \{<, >, =<, >=, =:=, =\backslash=\}$. Les deux derniers opérateurs représentent l'égalité et l'inégalité. Une requête « ?- $t_1 op t_2$. » réussit si t_1 et t_2 sont des expressions arithmétiques entièrement instanciées au moment de l'évaluation (c'est-à-dire qu'elles ne doivent plus contenir aucune variable) et si les valeurs z_1 et z_2 de t_1 et t_2 sont dans la relation op après l'infixe prédéfini les symboles de fonction ont été évalués. Les symboles de prédicat ci-dessus forcent donc une évaluation. Si t_1 ou t_2 n'est pas une expression arithmétique totalement instanciée, la requête n'échoue pas en cas d'échec, mais conduit plutôt à l'arrêt du programme. Les requêtes suivantes ont les résultats suivants :

— « ?- 1 < 2. » ou « ?- -2 < -1. » ou « ?- 1 * 1 < 1 + 1. » conduire au résultat vrai.
— « ?- 2 < 1. » ou « ?- 6 // 3 < 5-4. » résultat faux.
— « ?- a < 1. » ou « ?- X < 1. » a entraîné une erreur.

L'exigence selon laquelle toutes les variables doivent être instanciées au moment de l'exécution signifie que ces symboles de prédicat ne peuvent pas être utilisés pour instancier des variables via l'unification. Une requête comme « ?- X = := 2. » ne conduit pas à la substitution X = 2, mais à une erreur de programme. Pour cette raison, il existe un autre symbole prédéfini de prédicat. Une requête « ?- $t_1 ist_2$. » réussit si t_2 est une expression arithmétique entièrement instanciée au moment de l'évaluation, qui s'évalue à une valeur z_2, et si t_1 est unifiée avec z_2. Si t_2 n'est pas une expression arithmétique entièrement instanciée, la requête n'échoue pas en cas d'échec, mais conduit plutôt à l'arrêt du programme. Les requêtes suivantes ont les résultats suivants :

— « ?– 2 is 1 + 1. » ou « ?– 2 is 2. » conduire au résultat vrai.
— « ?- 1 + 1 is 2. » ou « ?- 1 + 1 is 1 + 1. » ou « ?- X + 1 is 1 + 1. » donne un résultat faux.
— « ?- X is 2. » ou « ?- X is 1 + 1. » conduit à la substitution X = 2
— « ?- X is 3 + 4, Y is X + 1. » conduit à la substitution X = 7 et Y = 8 (car au moment de l'évaluation, le X dans « Y is X + 1 » se voit attribuer une expression

arithmétique entièrement instanciée).
— « ?- X is X. », « ?- 2 is X. », « ?- X is 1. » ou « ?- Y is X + 1, X is 3 + 4. » a entraîné une erreur.

Il existe également un symbole X = X pour l'unification de termes arbitraires. Ce symbole est traité comme s'il était remplacé par le fait « X = X ». variables définies. Cela ne se limite donc pas aux expressions arithmétiques. Contrairement aux symboles spéciaux ci-dessus pour l'arithmétique (comme pour les autres symboles de prédicat auto-définis), il n'y a pas d'évaluation des symboles de fonction $+, -, *, //, **$, etc. Les requêtes suivantes ont les résultats suivants :

— « ?- a = a. » ou « ?- 2 = 2. » ou « ?- 1 + 1 = 1 + 1. » conduit au résultat vrai.
— « ?- 2 = 1 + 1. » ou « ?- 1 + 1 = 2. » donne résultat faux.
— « ?- X + 1 = 1 + 1. » ou « ?- 1 = X. » conduit à la substitution X = 1.
— « ?- X = 1 + 1. » conduit à la substitution X = 1 + 1.
— « ?- X = X. » conduit à la réponse vraie, c'est-à-dire à la substitution identique (vide).
— « ?- 1 + X = Y + 1. » conduit à une substitution avec X = 1 et Y = 1.
— « ?- X = 3 + 4, Y est X + 1. » conduit à la substitution X = 3 + 4 et Y = 8. À cet égard, nous avons différents types d'égalité :
— Valeurs égales « $t_1 =:= t_2$ », dans lesquelles t_1 et t_2 sont évalués et aucune unification n'aura lieu.
— Attribution de valeur « $t_1 ist_2$ », dans laquelle t_2 est évalué puis l'unification aura lieu.
— Égalité des termes « $t_1 = t_2$ », là où il n'y a pas d'évaluation, seule l'unification aura lieu.
— Égalité syntaxique « $t_1 == t_2$ », par laquelle on examine uniquement si t_1 et t_2 sont syntaxiquement identiques.

Par exemple, **X == X** conduit au résultat true, mais**X == Y** conduit au résultat false. De même, **f(a, X) == f(a, X)** conduit au résultat true, mais **f(a, X) == f(a, Y)** conduit au résultat false. L'exemple suivant montre comment écrire le programme d'addition depuis le début de la section à l'aide des fonctions arithmétiques prédéfinies (Alternativement, bien sûr, le programme « **add(X, Y, Z) :- Z is X + Y.** » serait également possible.) **add(X, 0, X).**
add(X, Y, Z) :- Y > 0, Y_1 **is Y-1, add(X,** Y_1**,** Z_1**), Z is** Z_1 **+ 1.** Comme prévu, la requête « **?- add(1,2, X).** » pour la substitution **X = 3**. L'avantage d'utiliser les fonctions et prédicats prédéfinis sur les nombres est une meilleure efficacité et une meilleure lisibilité. Un inconvénient est que la bidirectionnalité peut être perdue. La requête « **?- add(X, 2,3).** » conduit donc à une erreur de programme (car dans la requête « **Z is** Z_1 **+ 1** », alors $Z - 1$ n'est pas totalement instancié).
Le programme **add(X, 0, X).**
add(X, Y + 1, Z + 1) :- add (X, Y, Z).
Cependant, ce ne s'est pas comporté comme prévu. La requête « **?- add(1,2, X).** » conduit à faux, puisque 2 n'est pas unifié avec 0 ou **Y + 1**. La requête « **?- add(1,0 + 1, X).** » donne la substitution **X = 1 + 1**. Comme autres exemples typiques, nous montrons la programmation de la factorielle et l'algorithme de calcul du plus grand diviseur commun de deux nombres naturels.
fak(0,1).
fak(X,Y) :- X > 0, X_1 **is X-1, fak(**X_1**,**Y_1**), Y is X***Y_1**.**

ggT(X,0,X).
ggT(0,X,X).
ggT(X,Y,Z) :- X =< Y, X > 0, Y_1 **is Y-X, ggT(X,**Y_1**,Z).**

ggT(X,Y,Z) :- Y < X, Y > 0, X_1 **is X-Y, ggT(X1,Y,Z).**
La requête « **?- fak(3, X).** » entraîne la substitution **X = 6** et la requête « **?- ggT(28,36, X).** » donne la substitution **X = 4**.
Pour vérifier les types de termes, Prolog a prédéfini des prédicats tels que par exemple **number/1**. Ici, le **number(t)** est vrai si t est un nombre à ce point dans le temps (c'est-à-dire, les requêtes « **?- number(2)** » ou « **?- X is 1 + 1, number(X).** ») donne un résultat vrai , tandis que « **?- number(1 + 1).** » ou « **?- number(X).** » donne le résultat faux).

5.3 Liste

Pour représenter les listes sous forme de termes, on utilise généralement un symbole de fonction à zéro paramètres pour la liste vide et un symbole de prédicat à deux paramètres qui représente l'insertion d'un élément au début de la liste. Si ces symboles sont appelés $nil \in \Sigma_0$ et $contre \in \Sigma_2$, le programme Prolog suivant, par exemple, permet de calculer la longueur d'une liste (**length/2** est prédéfinie).
len(nil,0).
len(cons(X,X_s**),Y) :- len(**X_s**,**Y_1**), Y is** Y_1**+1.**
La requête « **?- len(cons(7,cons(3,nil)), X).** » donne donc la substitution X = 2. Si au lieu de nil le symbole de prédicat [] $\in \Sigma_0$ et au lieu de contre le symbole de prédicat $\in \Sigma_2$ prend, donc Prolog prend en charge des notations abrégées plus lisibles. Comme auparavant, vous pouvez maintenant écrire le programme suivant :
len([],0).
len(.(X,X_s**),Y) :- len(**X_s**,**Y_1**), Y is** Y_1**+1.**
Cependant, les abréviations suivantes sont également possibles :

— . (t1, t2) = [t1 | t2]
— . (t1, []) = [t1]
— . (t1, .(t2, .(t3, t))) = [t1, t2, t3 | t]
— . (t1, .(t2, .(t3, []))) = [t1,t2,t3] = [t1,t2 | [t3 | []]] = [t1 | [t2,t3 | []]] etc.

Ces abréviations sont considérées comme identiques à la notation avec « . » et « [] ».Par example :

— La requête « ?- [1,2] = [1 | [2]]. » ou « ?- [1,2] = .(1,[2]). » ou « ?- [1,2,3] = [1 | [2,3 | []]]. » ou « ?- .(1,.(2,[3])) = [1,2,3]. » ou « ?- .(1,2) = [1 | 2]. » retourne vrai.
— " ?- .(1,X) = [1,2,3]" donne la substitution X = [2,3].
— " ?- [X,[1 | X]] = [[2],Y]." donne la substitution X = [2], Y = [1,2].

On peut voir ça « . » est en réalité tout symbole de prédicat à deux paramètres qui peut donc également être utilisé pour représenter des arbres binaires. Donc .(.(1,2),.(3,4)) est en fait un arbre binaire, mais il peut être représenté par [[1 | 2] | [3 | 4]]. L'exemple de programme suivant examine si un élément est contenu dans une liste.

member(X,[X |]).
member(X,[$_{|}$ Y_s**]) :- member(X,**Y_s**).**

La requête « **?- member(X,[[a,b],1,[]]).** » donne la substitution **X = [a,b]**. Si vous entrez « ; » Si vous cherchez d'autres solutions, vous obtenez toujours **X = 1** et **X = []**. (Une autre entrée de « ; » donne la valeur false.) La requête « ? - membre (b, X). » recherche toutes les listes contenant b. Les solutions infiniment nombreuses **X = [b |** X_s**]** (toutes les listes avec le premier élément b), **X = [Y, b |** X_s**]** (toutes les listes avec le deuxième élément b), etc. sont obtenues l'une après l'autre.

Un autre prédicat classique est l'application pour la concaténation de listes (**append/3** est prédéfinie).

app([],Y_s**,** Y_s**).**
app([X | X_s**],** Y_s**,** $[X|Z_s]$**) :- app(**X_s**,** Y_s**,** Z_s**).**

La requête « ?- app ([1,2], [3,4,5], X_s). » donne la substitution X_s **= [1,2,3,4.5]**. La requête « ?- app (X_s, Y_s, [1,2,3])". » aboutit à la substitution X_s **= [],** Y_s **= [1,2,3]** et entrée répétée de « ; » génère trois autres solutions. Cependant, la requête « ?- app (X_s, [], Z_s). » a encore une fois un nombre infini de solutions.

5.4 Les opérateurs

La notation standard pour les termes et les atomes dans Prolog utilise la notation de préfixe, avec des arguments entre parenthèses. Par exemple, on écrit **p(X, f(a))** pour le prédicat ou symboles de fonction **p/2**, **f/1** et **a/0**. Ces symboles de prédicat et de fonction sont appelés foncteurs. Au lieu de cela, il est également possible d'utiliser des symboles de prédicat ou de fonction à deux arguments en notation infixe et des symboles de prédicat ou de fonction à un argument en notation préfixe ou postfixe (sans crochets pour les arguments). Pour ce faire, les symboles correspondants doivent être déclarés comme opérateurs.

L'avantage de ceci est une syntaxe plus conviviale avec une meilleure lisibilité. Cela vous rapproche de l'objectif de « programmation en langage naturel ».
Par exemple, **+** est déjà prédéfini comme opérateur. On peut donc écrire le terme **2 + 3**. Ceci est inséré dans le terme de Prolog **+(2,3)** converti, c'est-à-dire la requête « **?- 2 + 3 = + (2,3).** » renvoie vrai.

Des directives de la forme suivantes sont utilisées pour définir les opérateurs.

:- op(Precedence, Type, Nom (s)).

Il s'agit donc d'une clause avec un en-tête vide et le prédicat système op. Les opérateurs déclarés dans la directive peuvent alors être utilisés dans le programme après la position de leur déclaration. Les directives sont des requêtes qui sont traitées lorsque le programme est chargé, c'est-à-dire qu'au moment où la directive est chargée, une tentative est faite pour le prouver avec le programme qui a déjà été chargé. Les directives prédéfinies suivantes existent pour les opérateurs **+**, **-** et ***** :

:- op(500,yfx,[+,-]).
:- op(400,yfx,*).

Le dernier argument de op contient le symbole ou la liste des symboles qui sont déclarés ici comme opérateurs. Les précédents sont nécessaires pour exprimer la force avec laquelle chaque symbole de fonction se lie. Donc ***** se lie plus fortement que **+** et ceci est exprimé par le fait que ***** a une priorité plus petite. (Ainsi, une priorité plus petite signifie une liaison plus forte.) Le type détermine l'ordre de l'opérateur et de l'argument. Ici, **f** représente l'opérateur et **y** et **x** les arguments. Les types **xfx**,**yfx** et **xfy** sont disponibles pour les symboles infixes. Pour les symboles de préfixe, il existe des types **fx** et **fy** et pour les symboles de suffixe, il existe des types **xf** et **yf**.

Ici, la priorité de l'argument **x** doit vraiment être inférieure à la priorité de l'opérateur **f**. La priorité des arguments **y** doit être inférieure ou égale à la priorité de **f**. La

priorité d'un argument est la priorité de l'opérateur principal (et la priorité des foncteurs et des arguments entre crochets est 0). Dans le cas du type **yfx**, les arguments avec la même priorité élevée peuvent apparaître uniquement à gauche de l'opérateur. Cela signifie que **1 + 2 + 3** sera lu comme **(1 + 2) +3**. La requête aboutit donc à « **?- 1 + 2 + 3 = (1 + 2) +3.** » la réponse est vraie et la requête « **?- 1 + 2 + 3 = 1+ (2 + 3).** » renvoie false. Ces opérateurs sont associatifs à gauche. Il est donc également clair que **5-4-3** est lu comme **(5-4) -3**. La requête « **?- X is 5-4-3.** » délivre donc la substitution **X = -2**. De même, **xfy** déclare des opérateurs associatifs à droite et **xfx** déclare des opérateurs sans associativité.

Le terme **1 + 2 * 3 + 4** doit être lu comme **(1+ (2 * 3)) + 4**. La raison en est que chaque opérateur ne peut avoir que des arguments de priorité égale ou inférieure. Mais puisque * a une priorité inférieure à **+**, les deux termes **+** ne peuvent pas être les arguments de *. Bien entendu, les opérateurs peuvent également être surchargés. Il existe un autre opérateur prédéfini à un argument **-**.

:- op(200,fy,-).

L'expression **-2-3** signifie donc **(-2) -3**.

L'exemple suivant montre comment vous pouvez définir vos propres opérateurs afin de réaliser une forme simple de traitement du langage. Nous utilisons le verbe anglais « was » dans l'infixe. Cela ne devrait pas avoir d'associativité, car les phrases du type « Omar was students was excellent » n'ont aucun sens. Nous utilisons également le mot « of » dans la notation infixe, où de est destiné à s'associer à la droite. Donc « door of room of my house » signifie « door of (room of my house) ».
Donc, cela lie souvent plus fortement que ce qui devrait souvent avoir une priorité inférieure. Puis « Omar was student of Mohamed » signifie « Omar was (student of Mohamed) ». Enfin, nous avons introduit le mot « the ». Il s'agit d'un opérateur de préfixe sans associativité (car « the the student » n'a aucun sens). La priorité doit être inférieure à celle de « the ». Alors « the door of the room » signifie « (the door) of (the room) ».
Nous pouvons maintenant écrire le programme Prolog suivant :
:- op(300,xfx,was).
:- op(250,xfy,of).
:- op(200,fx,the).

mohamed was the son of the accountable of the firm.

Le fait « mohamed was the son of the accountable of the firm ». représente la formule atomique suivante avec le symbole de prédicat **was** et les symboles de fonction en **of** et **the**.

was(mohamed,of(the(son),of(the(accountable),the(firm))))

Nous pouvons maintenant poser les questions suivantes :
?- Who was the son of the accountable of the firm.
Who = mohamed
?- mohamed was What.
What = the son of the accountable of the firm
?- Who was the son of the accountable of What.
Who = mohamed
What = the firm

Chapitre 6

Conclusion

Nous sommes arrivés à la fin de ce cours ! Dans ce chapitre, nous ferons un résumé sur la programmation logique, notamment son principe, ses caractéristiques, les bases de la logique des prédicats, la résolution de base, l'unification et à la fin les programmes logiques. Tout d'abord, nous pourrions dire que la programmation favorise le développement de la réflexion formelle et la résolution de problème, elle offre aussi la motivation à explorer, à apprendre et à réfléchir. En plus, l'informatique nous a offert un moyen essentiel d'explorer et de comprendre notre monde à travers les différentes technologies générées. L'informatique est une invention qui nous a permis de voir le monde autrement, c'est un moyen pour explorer des entités, des relations, et des invariances jamais perçues auparavant par l'agent humain. Bien sûr, la liste pourrait inclure de nombreuses tâches passionnantes qui constituent des défis quotidiens. Ce qui est important, c'est que la programmation informatique moins en termes de construction d'outils qu'en tant que moyen de créer des modèles du monde réel. Par conséquent, notre compréhension accrue est alors reflétée dans la prochaine itération suivante de notre construction de modèle. C'est ainsi que naît la méthodologie de conception itérative, qu'elle soit utilisée par le programmeur individuel, ou comme c'est plus souvent le cas, au sein des communautés de collaboration de groupes de programmeurs, est une méthodologie critique pour comprendre nos domaines d'application, plus précisément l'intelligence artificielle. Dans l'avant-propos de ce cours, nous avons déclaré les raisons nécessaires de la connaissance des différents langages de programmation. Nous voulions démontrer l'importance des principaux langages utilisés dans ce domaine. Nous voulions également explorer la manière dont les problèmes que nous essayons de résoudre, les langages de programmation que nous créons pour nous aider à les résoudre, et la pratique de la programmation en IA se sont mutuellement façonnés. Prolog base son abstraction sur une théorie mathématique : dans ce cas, la logique formelle et les théorèmes de résolution. Dans ce cas, la logique formelle et la preuve de théorèmes par résolution. Cela permet à Prolog d'abstraire presque complètement la sémantique procédurale (le traitement de gauche à droite des buts et des mécanismes pragmatiques tels que la coupe sont des exceptions nécessaires. Le résultat est une sémantique déclarative qui permet aux programmeurs de voir les programmes comme des ensembles de contraintes sur les programmes (les solutions des problèmes). Aussi, parce que les grammaires prennent naturellement la forme de règles, Prolog n'a pas seulement prouvé sa valeur dans les applications de traitement du langage naturel, mais aussi comme outil de manipulation des langages formels, tels que les compilateurs ou les interpréteurs. Ce langage supporte des formes plus générales d'abstraction. L'organisation des programmes autour de types de données abstraits, de "paquets" de structures de données et d'opérations sur celles-ci, est un dispositif commun utilisé par les bons programmeurs, quel que soit le langage qu'ils utilisent. La complexité des problèmes d'IA exige clairement des formes puissantes de décomposition du

problème, mais la nature mal formée de nombreux problèmes de recherche défie les techniques courantes telles que la décomposition descendante. De même, l'utilisation de l'ordre des règles en Prolog, avec des instructions terminales non récursives précédant les règles récursives.

Prolog continue cette tradition avec son utilisation de la représentation logique et de la sémantique déclarative. La logique est l'outil classique de la pensée, donnant une base mathématique aux disciplines de la clarté, de la précision et de l'efficacité. Plus subtile est l'idée de sémantique déclarative, qui consiste à énoncer des contraintes sur la solution d'un problème indépendamment des étapes procédurales. Cela apporte un certain nombre d'avantages. Les programmes Prolog sont notoirement concis, puisque les mécanismes de calcul procédural sont abstraits de l'énoncé logique. Cette concision permet de formuler clairement les problèmes complexes rencontrés dans la programmation en IA. Les programmes de compréhension du langage naturel en sont l'exemple le plus évident, mais nous attirons également l'attention du lecteur sur la facilité relative d'écrire des méta-interprètes en Prolog. Comme la programmation a mûri en tant que discipline, nous en sommes également venus à reconnaître que les équipes écrivent généralement des logiciels complexes, plutôt qu'un seul génie travaillant de manière isolée. Ils sont parfaitement conscients que la complexité des problèmes auxquels l'informatique moderne fait l'exception du génie solitaire, plutôt que la règle. Notre objectif a été de donner aux étudiants une compréhension, non seulement de la puissance et la beauté du langage de programmation Prolog, mais aussi de la profondeur intellectuelle qu'implique leur maîtrise de ce langage. Cette maîtrise implique la syntaxe et la sémantique du langage, la compréhension de son utilisation, et la capacité à projeter les modèles de conception et d'implémentation qui définissent un langage bien écrit. Dans l'approche de cet objectif, nous nous sommes concentrés sur les problèmes communs à la programmation en intelligence artificielle, et nous avons raisonné à travers leurs solutions, en laissant l'utilisation du langage et les modèles d'organisation du programme. Nous espérons que notre méthode et son exécution dans ce cours ont aidé l'étudiant à comprendre les raisons profondes, les habitudes de pensée et de perception plus nuancées, derrière ces exemples. Il s'agit moins une question de connaissance que d'imagination.

Bibliographie

APT, Krzysztof R. (12 juil. 2001). « The Logic Programming Paradigm and Prolog ». In : *arXiv :cs/0107013*. arXiv : cs/0107013. URL : http://arxiv.org/abs/cs/0107013 (visité le 28/04/2021).

BRAMER, Max (2013). *Logic Programming with Prolog*. 2e éd. London : Springer-Verlag. ISBN : 978-1-4471-5486-0. DOI : 10.1007/978-1-4471-5487-7. URL : https://www.springer.com/gp/book/9781447154860 (visité le 28/04/2021).

BRATKO, Ivan (31 août 2011). *Prolog Programming for Artificial Intelligence*. 4th edition. Harlow, England ; New York : Pearson Education Canada. 696 p. ISBN : 978-0-321-41746-6.

CHEN, Yinong et Wei-tek TSAI (1er mai 2014). *Introduction to Programming Languages : Programming in C, C++, Scheme, Prolog, C#, and SOA*. Revised edizione. Kendall Hunt Pub Co. 353 p. ISBN : 978-1-4652-4700-1.

COLMERAUER, Alain (1990). « An Introduction to Prolog III ». In : *Computational Logic*. Sous la dir. de John W. LLOYD. ESPRIT Basic Research Series. Berlin, Heidelberg : Springer, p. 37-79. ISBN : 978-3-642-76274-1. DOI : 10.1007/978-3-642-76274-1_2.

LLOYD, J. W. (1984). *Foundations of Logic Programming*. Artificial Intelligence. Berlin Heidelberg : Springer-Verlag. ISBN : 978-3-642-96826-6. DOI : 10.1007/978-3-642-96826-6. URL : https://www.springer.com/gp/book/9783642968266 (visité le 17/05/2021).

LUGER, George F. et William A STUBBLEFIELD (2008). *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java for Artificial Intelligence : Structures and Strategies for Complex Problem Solving*. 6th. USA : Addison-Wesley Publishing Company. 464 p. ISBN : 978-0-13-607047-4.

NILSSON, Ulf et Jan MALUSZYNSKI (1er août 1995). *Logic, Programming and Prolog*. 2nd edition. Chichester ; New York : Wiley. 296 p. ISBN : 978-0-471-95996-0.

ROWE, N. (1988). « Artificial intelligence through Prolog ». In : DOI : 10.5860/choice.26-0350.

SHOHAM, Yoav (12 mai 2014). *Artificial Intelligence Techniques in Prolog*. Google-Books-ID : kySjBQAAQBAJ. Morgan Kaufmann. 348 p. ISBN : 978-1-4832-1449-8.

ZOMBORI, Zsolt, Josef URBAN et Chad E. BROWN (2020). « Prolog Technology Reinforcement Learning Prover ». In : *Automated Reasoning*. Sous la dir. de Nicolas PELTIER et Viorica SOFRONIE-STOKKERMANS. Lecture Notes in Computer Science. Cham : Springer International Publishing, p. 489-507. ISBN : 978-3-030-51054-1. DOI : 10.1007/978-3-030-51054-1_33.